USING THE FREERTOS REAL TIME KERNEL

A Practical Guide

Richard Barry

FREERTOS 实时内核 实用指南

这篇文章的英文原版我是在www.FreeRTOS.net上下载得到的。其实我并没有决定是否要在系统中使用FreeRTOS，虽然我想要的也仅仅是一个实时内核，当然更重要的是免费。之所以翻译这篇文章倒不是因为FreeRTOS有多么优秀，完全是因为这篇文章还不算太长。而且FreeRTOS.net仿佛致力于这个内核在国内的推广，也做了不少中文化的工作。所以我是打算利用工作之余，边看边译，到读完这篇文档，也就有个中文版了。如果FreeRTOS.net不弃的话，我倒是情愿放到这个网站上与大家共享。

另外，我本人很懒，没有翻译附录，而且译完正文后也没有做过任何检查。所以如果有任何问题，请不要骂我。

Zou Changjun
yisfx@126.com

第一章

任务管理

1.1 概览

[附录中提供了使用 FreeRTOS 源代码的实用信息]

小型多任务嵌入式系统简介

不同的多任务系统有不同的侧重点。以工作站和桌面电脑为例：

- 早期的处理器非常昂贵，所以那时的多任务用于实现在单处理器上支持多用户。这类系统中的调度算法侧重于让每个用户”公平共享”处理器时间。

- 随着处理器功能越来越强大，价格却更便宜，所以每个用户都可以独占一个或多个处理器。这类系统的调度算法则设计为让用户可以同时运行多个应用程序，而计算机也不会显得反应迟钝。例如某个用户可能同时运行了一个字处理程序，一个电子表格，一个邮件客户端和一个 WEB 浏览器，并且期望每个应用程序任何时候都能对输入有足够快的响应时间。

桌面电脑的输入处理可以归类为”软实时”。为了保证用户的最佳体验，计算机对每个输入的响应应当限定在一个恰当的时间范围——但是如果响应时间超出了限定范围，并不会让人觉得这台电脑无法使用。比如说，键盘操作必须在键按下后的某个时间内作出明显的提示。但如果按键提示超出了这个时间，会使得这个系统看起来响应太慢，而不致于说这台电脑不能使用。

仅仅从单处理器运行多线程这一点来说，实时嵌入式系统中的多任务与桌面电脑的多任务从概念上来讲是相似的。但实时嵌入式系统的侧重点却不同于桌面电脑——特别是当嵌入式系统期望提供”硬实时”行为的时候。

硬实时功能必须在给定的时间限制之内完成——如果无法做到即意味着整个系统的绝对失败。汽车的安全气囊触发机制就是一个硬实时功能的例子。安全气囊在撞击发生后给定时间限制内必须弹出。如果响应时间超出了这个时间限制，会使得驾驶员受到伤害，而这原本是可以避免的。

大多数嵌入式系统不仅能满足硬实时要求，也能满足软实时要求。

术语说明

在 FreeRTOS 中，每个执行线程都被称为"任务"。在嵌入式社区中，对此并没有一个公允的术语，但我更喜欢用"任务"而不是"线程"，因为从以前的经验来看，线程具有更多的特定含义。

本章的目的是让读者充分了解：

- 在应用程序中，FreeRTOS 如何为各任务分配处理时间。
- 在任意给定时刻，FreeRTOS 如何选择任务投入运行。
- 任务优先级如何影响系统行为。
- 任务存在哪些状态。

此外，还期望能够让读者解：

- 如何实现一个任务。
- 如何创建一个或多个任务的实例。
- 如何使用任务参数。
- 如何改变一个已创建任务的优先级。
- 如何删除任务。
- 如何实现周期性处理。
- 空闲任务何时运行，可以用来干什么。

本章所介绍的概念是理解如何使用 FreeRTOS 的基础，也是理解基于 FreeRTOS 的应用程序行为方式的基础——因此，本章也是这本书中最为详尽的一章。

1.2 任务函数

任务是由 C 语言函数实现的。唯一特别的只是任务的函数原型，其必须返回 void，而且带有一个 void 指针参数。其函数原型参见**程序清单 1**。

```
void ATaskFunction( void *pvParameters );
```

程序清单 1 任务函数原型

每个任务都是在自己权限范围内的一个小程序。其具有程序入口，通常会运行在一个死循环中， 也不会退出。一个典型的任务结构如**程序清单 2** 所示。

FreeRTOS 任务不允许以任何方式从实现函数中返回——它们绝不能有一条”return”语句，也不能执行到函数末尾。如果一个任务不再需要，可以显式地将其删除。这也在**程序清单 2** 展现。

一个任务函数可以用来创建若干个任务——创建出的任务均是独立的执行实例，拥有属于自己的栈空间，以及属于自己的自动变量(栈变量)，即任务函数本身定义的变量。

```
void ATaskFunction( void *pvParameters )
{
/* 可以像普通函数一样定义变量。用这个函数创建的每个任务实例都有一个属于自己的iVarialbleExample变量。但如果iVariableExample被定义为static，这一点则不成立 – 这种情况下只存在一个变量，所有的任务实例将会共享这个变量。 */
int iVariableExample = 0;

    /* 任务通常实现在一个死循环中。 */
    for( ;; )
    {
        /* 完成任务功能的代码将放在这里。 */
    }

    /* 如果任务的具体实现会跳出上面的死循环，则此任务必须在函数运行完之前删除。传入NULL参数表示删除的是当前任务 */
    vTaskDelete( NULL );
}
```

程序清单 2 典型的任务函数结构

1.3 顶层任务状态

应用程序可以包含多个任务。如果运行应用程序的微控制器只有一个核(core)，那么在任意给定时间，实际上只会有一个任务被执行。这就意味着一个任务可以有一个或两个状态，即运行状态和非运行状态。我们先考虑这种最简单的模型——但请牢记这其实是过于简单，我们稍后将会看到非运行状态实际上又可划分为若干个子状态。

当某个任务处于运行态时，处理器就正在执行它的代码。当一个任务处于非运行态时，该任务进行休眠，它的所有状态都被妥善保存，以便在下一次调试器决定让它进入运行态时可以恢复执行。当任务恢复执行时，其将精确地从离开运行态时正准备执行的那一条指令开始执行。

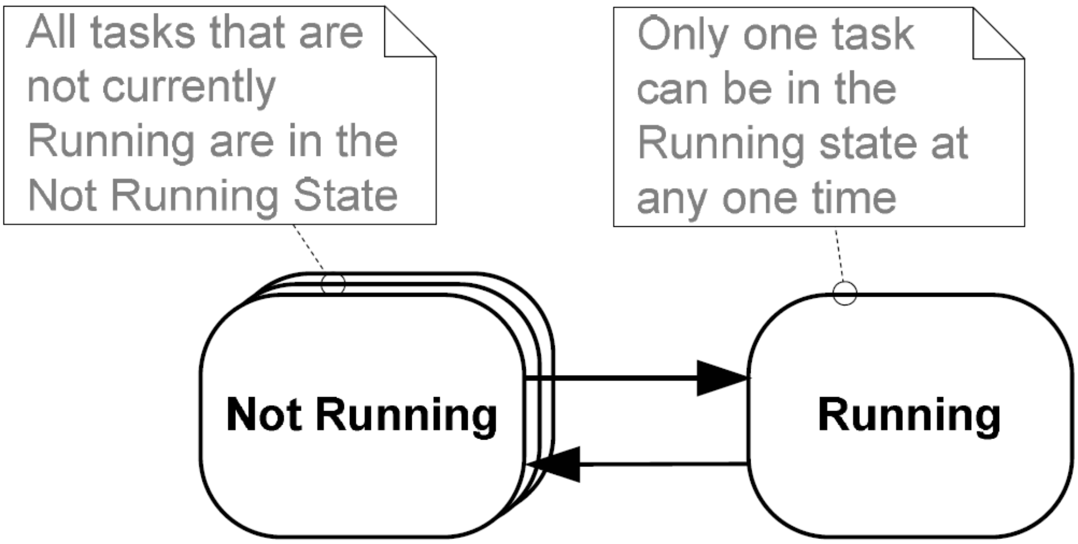


图 1　顶层任务状态及状态转移

任务从非运行态转移到运行态被称为"切换入或切入(switched in)"或"交换入(swapped in)"。相反，任务从运行态转移到非运行态被称为"切换出或切出(switched out)"或"交换出(swapped out)"。FreeRTOS 的调度器是能让任务切入切出的唯一实体。

1.4 创建任务

xTaskCreate() API 函数

创建任务使用 FreeRTOS 的 API 函数 xTaskCreate()。这可能是所有 API 函数中最复杂的函数，但不幸的是这也是我们第一个遇到的 API 函数。但我们必须首先掌控任务，因为它们是多任务系统中最基本的组件。本书中的所有示例程序都会用到 xTaskCreate()，所以会有大量的例子可以参考。

附录 5：描述用到的数据类型和命名约定。

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask );
```

程序清单 3　xTaskCreate() API 函数原型

表 1　xTaskCreate()参数与返回值

参数名	描述
pvTaskCode	任务只是永不退出的 C 函数，实现常通常是一个死循环。参数 pvTaskCode 只一个指向任务的实现函数的指针(效果上仅仅是函数名)。
pcName	具有描述性的任务名。这个参数不会被 FreeRTOS 使用。其只是单纯地用于辅助调试。识别一个具有可读性的名字总是比通过句柄来识别容易得多。 应用程序可以通过定义常量 config_MAX_TASK_NAME_LEN 来定义任务名的最大长度——包括'\0'结束符。如果传入的字符串长度超过了这个最大值，字符串将会自动被截断。

usStackDepth 当任务创建时，内核会分为每个任务分配属于任务自己的唯一状态。usStackDepth 值用于告诉内核为它分配多大的栈空间。

这个值指定的是栈空间可以保存多少个字(word)，而不是多少个字节(byte)。比如说，如果是 32 位宽的栈空间，传入的 usStackDepth 值为 100，则将会分配 400 字节的栈空间(100 * 4bytes)。栈深度乘以栈宽度的结果千万不能超过一个 size_t 类型变量所能表达的最大值。

应用程序通过定义常量 configMINIMAL_STACK_SIZE 来决定空闲任务任用的栈空间大小。在 FreeRTOS 为微控制器架构提供的 Demo 应用程序中，赋予此常量的值是对所有任务的最小建议值。如果你的任务会使用大量栈空间，那么你应当赋予一个更大的值。

没有任何简单的方法可以决定一个任务到底需要多大的栈空间。计算出来虽然是可能的，但大多数用户会先简单地赋予一个自认为合理的值，然后利用 FreeRTOS 提供的特性来确证分配的空间既不欠缺也不浪费。第六章包括了一些信息，可以知道如何去查询任务使用了多少栈空间。

pvParameters 任务函数接受一个指向 void 的指针(void*)。pvParameters 的值即是传递到任务中的值。这篇文档中的一些范例程序将会示范这个参数可以如何使用。

uxPriority 指定任务执行的优先级。优先级的取值范围可以从最低优先级 0 到最高优先级(configMAX_PRIORITIES – 1)。

configMAX_PRIORITIES 是一个由用户定义的常量。优先级号并没有上限(除了受限于采用的数据类型和系统的有效内存空间)，但最好使用实际需要的最小数值以避免内存浪费。如果 uxPriority 的值超过了(configMAX_PRIORITIES – 1)，将会导致实际赋给任务的优先级被自动封顶到最大合法值。

pxCreatedTask	pxCreatedTask 用于传出任务的句柄。这个句柄将在 API 调用中对该创建出来的任务进行引用，比如改变任务优先级，或者删除任务。 如果应用程序中不会用到这个任务的句柄，则 pxCreatedTask 可以被设为 NULL。
返回值	有两个可能的返回值： 1. pdTRUE 表明任务创建成功。 2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY 由于内存堆空间不足，FreeRTOS 无法分配足够的空间来保存任务结构数据和任务栈，因此无法创建任务。 第五章将提供更多有关内存管理方面的信息。

例 1. 创建任务

附录 1：包含一些关于示例程序生成工具的信息。

本例演示了创建并启动两个任务的必要步骤。这两个任务只是周期性地打印输出字符串，采用原始的空循环方式来产生周期延迟。两者在创建时指定了相同的优先级，并且在实现上除输出的字符串外完全一样——**程序清单** 4 和**程序清单** 5 是这两个任务对应的实现代码。

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /* 和大多数任务一样，该任务处于一个死循环中。 */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* 延迟，以产生一个周期 */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* 这个空循环是最原始的延迟实现方式。在循环中不做任何事情。后面的示例程序将采用
            delay/sleep函数代替这个原始空循环。 */
        }
    }
}
```

程序清单4　例1中的第一个任务实现代码

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile unsigned long ul;

    /* 和大多数任务一样，该任务处于一个死循环中。 */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* 延迟，以产生一个周期 */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* 这个空循环是最原始的延迟实现方式。在循环中不做任何事情。后面的示例程序将采用
            delay/sleep函数代替这个原始空循环。 */
        }
    }
}
```

程序清单 5　例 1 中的第二个任务实现代码

main()函数只是简单地创建这两个任务，然后启动调度器——具体实现代码参见程序清单 6。

```
int main( void )
{
    /* 创建第一个任务。需要说明的是一个实用的应用程序中应当检测函数xTaskCreate()的返回值，以确保任
    务创建成功。 */
    xTaskCreate( vTask1,      /* 指向任务函数的指针 */
                 "Task 1",    /* 任务的文本名字，只会在调试中用到 */
                 1000,        /* 栈深度 – 大多数小型微控制器会使用的值会比此值小得多 */
                 NULL,        /* 没有任务参数 */
                 1,           /* 此任务运行在优先级1上. */
                 NULL );      /* 不会用到任务句柄 */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* 启动调度器，任务开始执行 */
    vTaskStartScheduler();

    /* 如果一切正常，main()函数不应该会执行到这里。但如果执行到这里，很可能是内存堆空间不足导致空闲
    任务无法创建。第五章有讲述更多关于内存管理方面的信息 */
    for( ;; );
}
```

程序清单 6　启动例 1 中的任务

本例的运行输出如图 2 所示

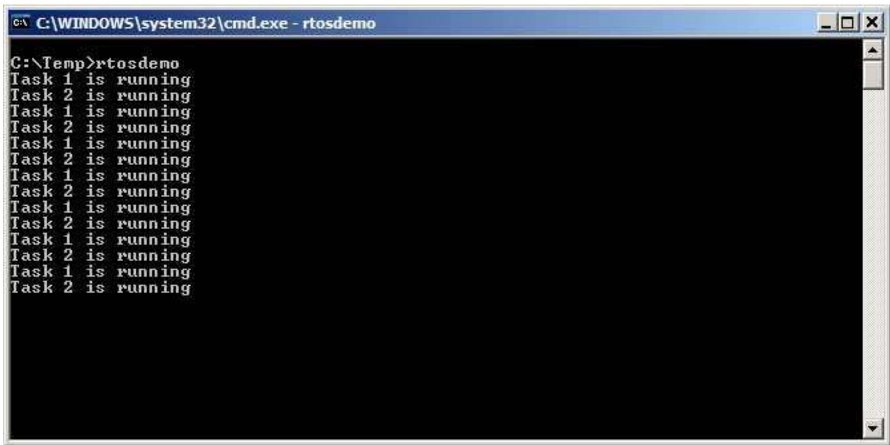


图 2 例 1 的运行输出

图 2 中看到两个任务在同时运行，但实际上这两个任务运行在同一个处理器上，所以不可能会同时运行。事实上这两个任务都迅速地进入与退出运行态。由于这两个任务运行在同一个处理器上，所以会平等共享处理器时间。真实的执行流程所图 3 所示。

图 3 中底部的箭头表示从 t1 起始的运行时刻。彩色的线段表示在每个时间点上正在运行的任务——比如 t1 与 t2 之间运行的是任务 1。

在任何时刻只可能有一个任务处于运行态。所以一个任务进入运行态后(切入)，另一个任务就会进入非运行态(切出)。

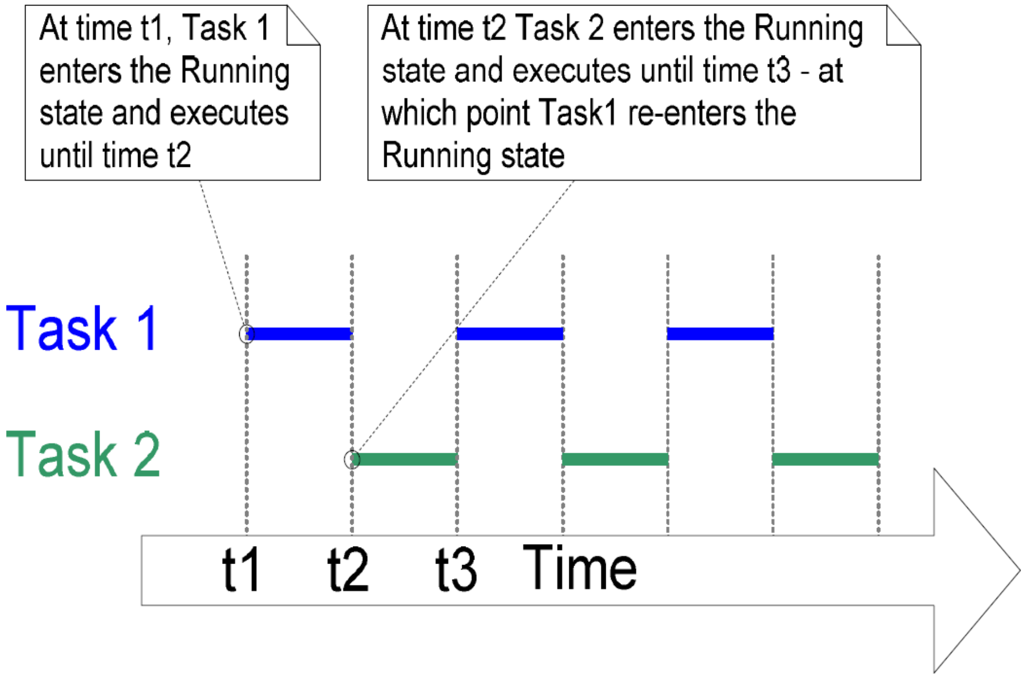


图 3　例 1 的实际执行流程

例 1 中，main()函数在启动调度器之前先完成两个任务的创建。当然也可以从一个任务中创建另一个任务。我们可以先在 main()中创建任务 1，然后在任务 1 中创建任务 2。如果我们需要这样做，则任务 1 代码就应当修改成**程序清单 7** 所示的样子。这样，在调度器启动之前，任务 2 还没有被创建，但是整个程序运行的输出结果还是相同的。

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /*  如果已经执行到本任务的代码，表明调度器已经启动。在进入死循环之前创建另一个任务。 */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

程序清单 7　在一个任务中创建另一个任务 —— 在调度器启动之后

例 2. 使用任务参数

例 1 中创建的两个任务几乎完全相同，唯一的区别就是打印输出的字符串。这种重复性可以通过创建同一个任务代码的两个实例来去除。这时任务参数就可以用来传递各自打印输出的字符串。

程序清单 8 包含了**例 2** 中用到的唯一一个任务函数代码(vTaskFunction)。这一个任务函数代替了**例 1** 中的两个任务函数(vTask1 与 vTask2)。这个函数的任务参数被强制转化为 char*以得到任务需要打印输出的字符串。

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile unsigned long ul;

    /* 需要打印输出的字符串从入口参数传入。强制转换为字符指针。 */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

程序清单 **8**　例 **2** 中用于创建两个任务实例的任务函数

尽管现在只有一个任务实现代码(vTaskFunction)，但是可以创建多个任务实例。每个任务实例都可以在 FreeRTOS 调度器的控制下独运行。

传递给 API 函数 xTaskCreate()的参数 pvPrameters 用于传入字符串文本。如程序清单 **9** 所示。

```
/* 定义将要通过任务参数传递的字符串。定义为const，且不是在栈空间上，以保证任务执行时也有效。 */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\t\n";
int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate( vTaskFunction,          /* 指向任务函数的指针. */
                 "Task 1",               /* 任务名. */
                 1000,                   /* 栈深度. */
                 (void*)pcTextForTask1, /* 通过任务参数传入需要打印输出的文本. */
                 1,                      /* 此任务运行在优先级1上. */
                 NULL );                 /* 不会用到此任务的句柄. */

    /* 同样的方法创建另一个任务。至此，由相同的任务代码(vTaskFunction)创建了多个任务，仅仅是传入
    的参数不同。同一个任务创建了两个实例。 */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

    /* Start the scheduler so our tasks start executing. */
    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    CHAPTER 5 provides more information on memory management. */
    for( ;; );
}
```

程序清单 **9** 例 **2** 中的 **main()**函数实现代码

例 **2** 的运行输出结果与例 **1** 完全一样，参见图 **2**。

1.5 任务优先级

xTaskCreate() API 函数的参数 uxPriority 为创建的任务赋予了一个初始优先级。这个优先级可以在调度器启动后调用 vTaskPrioritySet() API 函数进行修改。

应用程序在文件 FreeRTOSConfig.h 中设定的编译时配置常量 configMAX_PRIORITIES 的值，即是最多可具有的优先级数目。FreeRTOS 本身并没有限定这个常量的最大值，但这个值越大，则内核花销的内存空间就越多。所以总是建议将此常量设为能够用到的最小值。

对于如何为任务指定优先级，FreeRTOS 并没有强加任何限制。任意数量的任务可以共享同一个优先级——以保证最大设计弹性。当然，如果需要的话，你也可以为每个任务指定唯一的优先级(就如同某些调度算法的要求一样)，但这不是强制要求的。

低优先级号表示任务的优先级低，优先级号 0 表示最低优先级。有效的优先级号范围从 0 到(configMAX_PRIORITES – 1)。

调度器保证总是在所有可运行的任务中选择具有最高优先级的任务，并使其进入运行态。如果被选中的优先级上具有不止一个任务，调度器会让这些任务轮流执行。这种行为方式在之前的例子中可以明显看出来。两个测试任务被创建在同一个优先级上，并且一直是可运行的。所以每个任务都执行一个"时间片"，任务在时间片起始时刻进入运行态，在时间片结束时刻又退出运行态。图 **3** 中 t1 与 t2 之间的时段就等于一个时间片。

要能够选择下一个运行的任务，调度器需要在每个时间片的结束时刻运行自己本身。一个称为心跳(tick，有些地方被称为时钟滴答，本文中一律称为时钟心跳)中断的周期性中断用于此目的。时间片的长度通过心跳中断的频率进行设定，心跳中断频率由 FreeRTOSConfig.h 中的编译时配置常量 configTICK_RATE_HZ 进行配置。比如说，如果 configTICK_RATE_HZ 设为 100(HZ)，则时间片长度为 10ms。可以将图 **3** 进行扩展，将调度器本身的执行时间在整个执行流程中体现出来。请参见图 **4**。

需要说明的是，FreeRTOS API 函数调用中指定的时间总是以心跳中断为单位（通常的提法为心跳"ticks"）。常量 portTICK_RATE_MS 用于将以心跳为单位的时间值转化为以毫秒为单位的时间值。有效精度依赖于系统心跳频率。

心跳计数(tick count)值表示的是从调度器启动开始，心跳中断的总数，并假定心跳

计数器不会溢出。用户程序在指定延迟周期时不必考虑心跳计数溢出问题，因为时间连贯性在内核中进行管理。

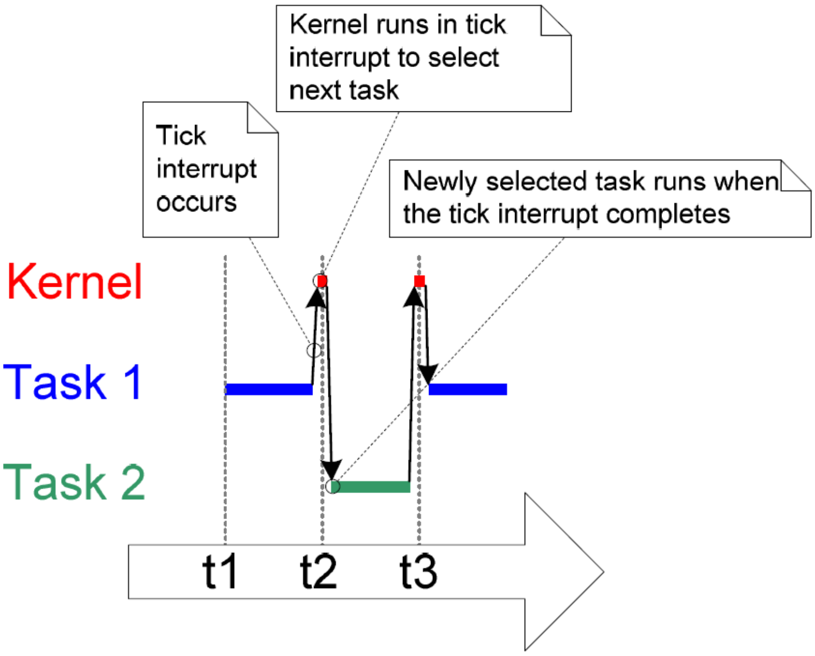


图 4 对执行流程进行扩展以显示心跳中断的执行

图 4 中红色的线段表时内核本身在运行。黑色箭头表示任务到中断，中断再到另一个任务的执行顺序。

例 3. 优先级实验

调度器总是在可运行的任务中，选择具有最高优级级的任务，并使其进入运行态。到目前为止的示例程序中，两个任务都创建在相同的优先级上。所以这两个任务轮番进入和退出运行态。本例将改变**例 2** 其中一个任务的优先级，看一下倒底会发生什么。现在第一个任务创建在优先级 1 上，而另一个任务创建在优先级 2 上。创建这两个任务的代码参见**程序清单 10**。这两个任务的实现函数没有任何改动，还是通过空循环产生延迟来周期性打印输出字符串。

```
/* 定义将要通过任务参数传递的字符串。定义为const，且不是在栈空间上，以保证任务执行时也有效。 */
static const char *pcTextForTask1 = “Task 1 is running\r\n”;
static const char *pcTextForTask2 = “Task 2 is running\t\n”;

int main( void )
{
    /* 第一个任务创建在优先级1上。优先级是倒数第二个参数。 */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* 第二个任务创建在优先级2上。 */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    return 0;
}
```

程序清单 **10** 两个任务创建在不同的优先级上

图 **5** 是例 **3** 的运行结果。

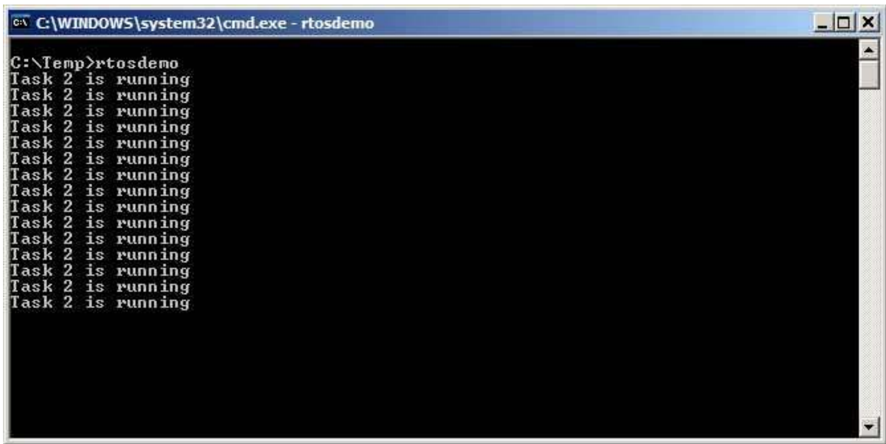


图 **5** 两个测试任务运行在不同的优先级上

调度器总是选择具有最高优先级的可运行任务来执行。任务 2 的优先级比任务 1 高，并且总是可运行，因此任务 2 是唯一一个一直处于运行态的任务。而任务 1 不可能进入运行态，所以不可能输出字符串。这种情况我们称为任务 1 的执行时间被任务 2”饿死(starved)”了。

任务 2 之所以总是可运行，是因为其不会等待任何事情——它要么在空循环里打转，要么往终端打印字符串。

图 6 展现了**例 3** 的执行流程。

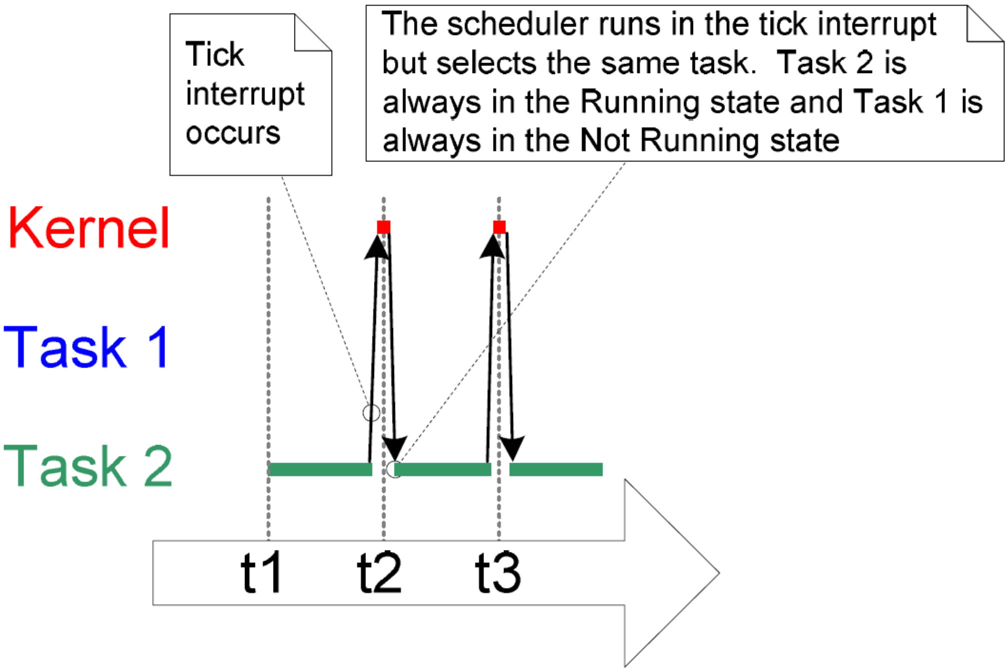


图 6 当一个任务优先比另一个高时的执行流程

1.6 扩充“非运行态”

到目前为止所有用到的示例中，创建的每个任务都只顾不停地处理自己的事情而没有其它任何事情需要等待——由于它们不需要等待所以总是能够进入运行态。这种”不停处理”类型的任务限制了其有用性，因为它们只可能被创建在最低优先级上。如何它们运行在其它任何优先级上，那么比它们优先级更低的任务将永远没有执行的机会。

为了使我们的任务切实有用，我们需要通过某种方式来进行事件驱动。一个事件驱动任务只会在事件发生后触发工作(处理)，而在事件没有发生时是不能进入运行态的。调度器总是选择所有**能够**进入运行态的任务中具有最高优先级的任务。一个高优先级但不能够运行的任务意味着不会被调度器选中，而代之以另一个优先级虽然更低但能够运行的任务。因此，采用事件驱动任务的意义就在于任务可以被创建在许多不同的优先级上，并且最高优先级任务不会把所有的低优先级任务饿死。

阻塞状态

如果一个任务正在等待某个事件，则称这个任务处于”阻塞态(blocked)”。阻塞态是非运行态的一个子状态。

任务可以进入阻塞态以等待以下两种不同类型的事件：

1. 定时(时间相关)事件——这类事件可以是延迟到期或是绝对时间到点。比如说某个任务可以进入阻塞态以延迟 10ms。
2. 同步事件——源于其它任务或中断的事件。比如说，某个任务可以进入阻塞态以等待队列中有数据到来。同步事件囊括了所有板级范围内的事件类型。

FreeRTOS 的队列，二值信号量，计数信号量，互斥信号量(recursive semaphore, 递归信号量，本文一律称为互斥信号量，因为其主要用于实现互斥访问)和互斥量都可以用来实现同步事件。第二章和第三章涵盖了有关这些的详细内容。

任务可以在进入阻塞态以等待同步事件时指定一个等待超时时间，这样可以有效地实现阻塞状态下同时等待两种类型的事件。比如说，某个任务可以等待队列中有数据到来，但最多只等 10ms。如果 10ms 内有数据到来，或是 10ms 过去了还没有数据到来，这两种情况下该任务都将退出阻塞态。

挂起状态

“挂起(suspended)”也是非运行状态的子状态。处于挂起状态的任务对调度器而言是不可见的。让一个任务进入挂起状态的唯一办法就是调用 vTaskSuspend() API 函数；而把一个挂起状态的任务唤醒的唯一途径就是调用 vTaskResume() 或 vTaskResumeFromISR() API 函数。大多数应用程序中都不会用到挂起状态。

就绪状态

如果任务处于非运行状态，但既没有阻塞也没有挂起，则这个任务处于就绪(ready，准备或就绪)状态。处于就绪态的任务能够被运行，但只是”准备(ready)”运行，而当前尚未运行。

完整的状态转移图

图 7 对之前那个过于简单的状态图进行了扩充，包含了本节描述的非运行状态的子状态。目前为止所有用到的示例程序中创建的任务都还没有用到阻塞状态和挂起状态，仅仅是在就绪状态和运行状态之间转移——**图 7** 中以粗线进行醒目提示。

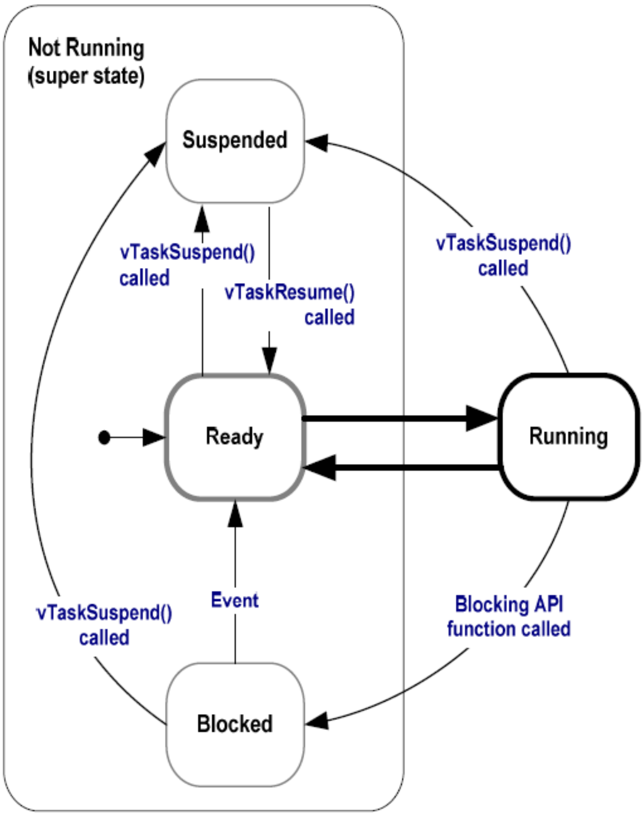


图 7 完整的任务状态机

例 4. 利用阻塞态实现延迟

之前的示例中所有创建的任务都是"周期性"的——它们延迟一个周期时间，打印输出字符串，再一次延迟，如此周而复始。而产生延迟的方法也相当原始地使用了空循环——不停地查询并递增一个循环计数直至计到某个指定值。例 3 明确的指出了这种方法的缺点。一直保持在运行态中执行空循环，可能将其它任务饿死。

其实以任何方式的查询都不仅仅只是低效，还有各种其它方面的缺点。在查询过程中，任务实际上并没有做任何有意义的事情，但它依然会耗尽所有处理时间，对处理器周期造成浪费。**例 4** 通过调用 vTaskDelay() API 函数来代替空循环，对这种"不良行为"进行纠正。vTaskDelay()的函数原型见**程序清单 11**，而新的任务实现见**程序清单 12**。

```
void vTaskDelay( portTickType xTicksToDelay );
```

程序清单 11 vTaskDelay() API 函数原型

表 2 vTaskDelay()参数

参数名	描述
xTicksToDelay	延迟多少个心跳周期。调用该延迟函数的任务将进入阻塞态，经延迟指定的心跳周期数后，再转移到就绪态。 举个例子，当某个任务调用 vTaskDelay(100)时，心跳计数值为 10,000，则该任务将保持在阻塞态，直到心跳计数计到 10,100。 常数 portTICK_RATE_MS 可以用来将以毫秒为单位的时间值转换为以心跳周期为单位的时间值。

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;

    /* The string to print out is passed in via the parameter. Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* 延迟一个循环周期。调用vTaskDelay()以让任务在延迟期间保持在阻塞态。延迟时间以心跳周期为
        单位，常量portTICK_RATE_MS可以用来在毫秒和心跳周期之间相换转换。本例设定250毫秒的循环周
        期。 */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

程序清单 12　调用 vTaskDelay()来代替空循环实现延迟

尽管两个任务实例还是创建在不同的优先级上，但现在两个任务都可以得到执行。**例 4** 的运行输出结果参见**图 8**。



图 9 所示的执行流程可以解释为什么此时不同优先级的两个任务竟然都可以得到执行。图中为了简便，忽略了内核自身的执行时间。

空闲任务是在调度器启动时自动创建的，以保证至少有一个任务可运行(至少有一个任务处于就绪态)。本章第 7 节会对空闲任务进行更详细的描述。

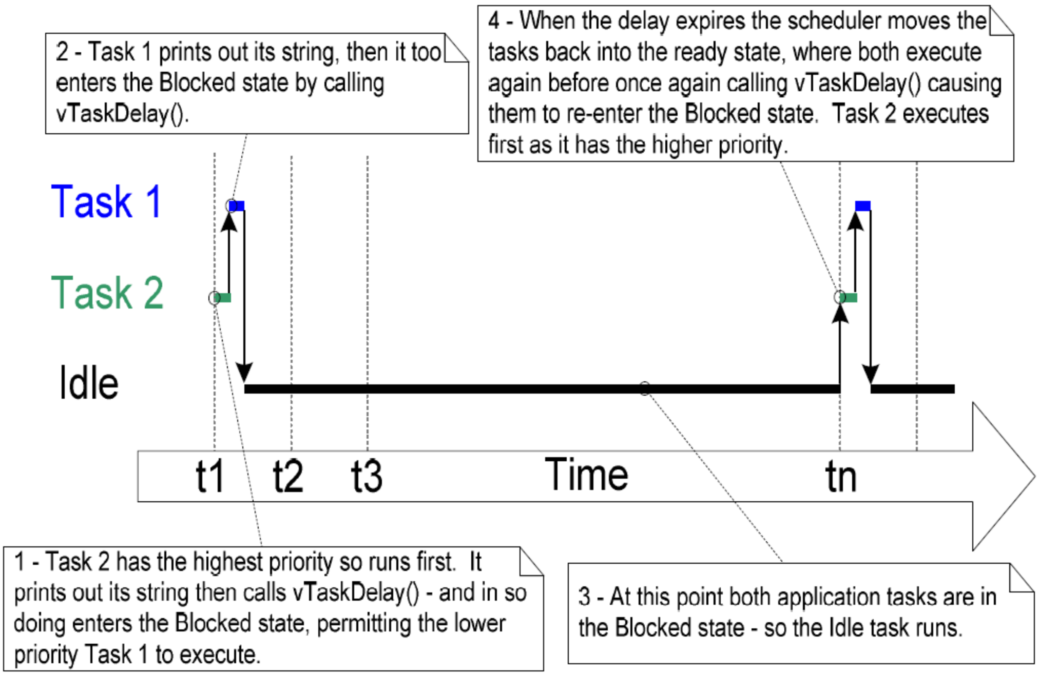


图 9 用 **vTaskDelay()**代替空循环后的执行流程

本例是只改变了两个任务的实现方式，并没有改变其功能。对比**图 9** 与**图 4** 可以清晰地看到本例以更有效的方式实现了任务的功能。

图 4 展现的是当任务采用空循环进行延迟时的执行流程——结果就是任务总是可运行并占用了大量的机器周期。从**图 9** 中的执行流程中可以看到，任务在整个延迟周期内都处于阻塞态，只在完成实际工作的时候才占用处理器时间(本例中任务的实际工作只是简单地打印输出一条信息)。

在**图 9** 所示的情形中，任务离开阻塞态后，仅仅执行了一个心跳周期的一个片段，然后又再次进入阻塞态。所以大多数时间都没有一个应用任务可运行(即没有应用任务处于就绪态)，因此没有应用任务可以被选择进入运行态。这种情况下，空闲任务得以执行。空间任务可以获得的执行时间量，是系统处理能力裕量的一个度量指标。

图 10 中的粗线条表示例 4 中任务的状态转移过程。现在每个任务在返回就绪态之前，都会经过阻塞状态。

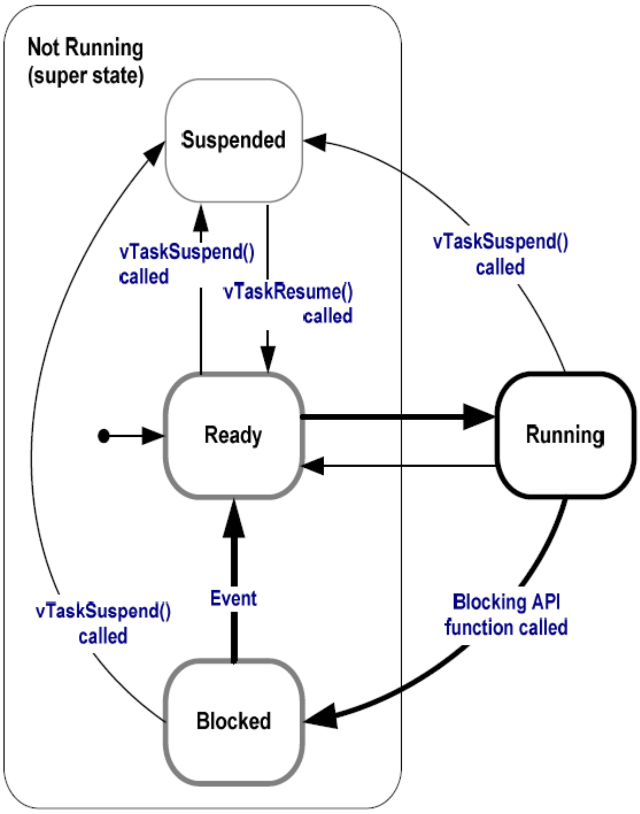


图 10 粗线条表示例 4 中的状态转移过程

vTaskDelayUntil() API 函数

vTaskDelayUntil()类似于 vTaskDelay()。和范例中演示的一样，函数 vTaskDelay()的参数用来指定任务在调用 vTaskDelay()到切出阻塞态整个过程包含多少个心跳周期。任务保持在阻塞态的时间量由 vTaskDelay()的入口参数指定，但任务离开阻塞态的时刻实际上是相对于 vTaskDelay()被调用那一刻的。vTaskDelayUntil()的参数就是用来指定任务离开阻塞态进入就绪态那一刻的精确心跳计数值。API 函数 vTaskDelayUntil()可以用于实现一个固定执行周期的需求(当你需要让你的任务以固定频率周期性执行的时候)。由于调用此函数的任务解除阻塞的时间是绝对时刻，比起相对于调用时刻的相对时间更精确(即比调用 vTaskDelay()可以实现更精确的周期性)。

```
void vTaskDelayUntil( portTickType * pxPreviousWakeTime, portTickType xTimeIncrement );
```

程序清单 13 vTaskDelayUntil() API 函数原型

表 3 vTaskDelayUntil()参数

参数名	描述
pxPreviousWakeTime	此参数命名时假定 vTaskDelayUntil()用于实现某个任务以固定频率周期性执行。这种情况下 pxPreviousWakeTime 保存了任务上一次离开阻塞态(被唤醒)的时刻。这个时刻被用作一个参考点来计算该任务下一次离开阻塞态的时刻。 pxPreviousWakeTime 指向的变量值会在 API 函数 vTaskDelayUntil()调用过程中自动更新，应用程序除了该变量第一次初始化外，通常都不要修改它的值。程序清单 **14** 展示了这个参数的使用方法。
xTimeIncrement	此参数命名时同样是假定 vTaskDelayUntil()用于实现某个任务以固定频率周期性执行 —— 这个频率就是由 xTimeIncrement 指定的。 xTimeIncrement 的单位是心跳周期，可以使用常量 portTICK_RATE_MS 将毫秒转换为心跳周期。

例 5. 转换示例任务使用 vTaskDelayUntil()

例 4 中的两个任务是周期性任务，但是使用 vTaskDelay()无法保证它们具有固定的执行频率，因为这两个任务退出阻塞态的时刻相对于调用 vTaskDelay()的时刻。通过调用 vTaskDelayUntil()代替 vTaskDelay()，把这两个任务进行转换，以解决这个潜在的问题。

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
portTickType xLastWakeTime;

    /* The string to print out is passed in via the parameter. Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* 变量xLastWakeTime需要被初始化为当前心跳计数值。说明一下，这是该变量唯一一次被显式赋值。之后，
    xLastWakeTime将在函数vTaskDelayUntil()中自动更新。 */
    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* 本任务将精确的以250毫秒为周期执行。同vTaskDelay()函数一样，时间值是以心跳周期为单位的，
        可以使用常量portTICK_RATE_MS将毫秒转换为心跳周期。变量xLastWakeTime会在
        vTaskDelayUntil()中自动更新，因此不需要应用程序进行显示更新。 */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

程序清单 14　使用 vTaskDelayUntil()实现示例任务

例 5 的运行输出与**例 4** 完全相同，请参考**图 8**。

例 6. 合并阻塞与非阻塞任务

之前的范例分别测试了任务以查询方式和阻塞方式工作的系统行为。本例通过合并这两种方案的执行流程，再次实现具有既定预期的系统行为。

- 在优先级 1 上创建两个任务。这两个任务只是不停地打印输出字符串，然它什么事情也不做。

  这两个任务没有调用任何可能导致它们进入阻塞态的 API 函数，所以这两个任务要么处于就绪态，要么处于运行态。具有这种性质的任务被称为"不停处理(或持续处理，continuous processing)"任务，因为它们总是有事情要做，虽然在本例中的它们做的事情没什么意义。持续处理任务的源代码参见**程序清单 15**。

- 第三个任务创建在优先级 2 上，高于另外两个任务的优先级。这个任务虽然也是打印输出字符串，但它是周期性的，所以调用了 vTaskDelayUntil()，在每两次打印之间让自己处于阻塞态。

  周期性任务的实现代码参见**程序清单 16**。

```
void vContinuousProcessingTask( void *pvParameters )
{
char *pcTaskName;

    /* 打印输出的字符串由任务参数传入，强制转换为char* */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* 打印输出任务名，无阻塞，也无延迟。 */
        vPrintString( pcTaskName );
    }
}
```

程序清单 15 例6中持续处理任务的实现代码

```
void vPeriodicTask( void *pvParameters )
{
portTickType xLastWakeTime;

    /* 初始化xLastWakeTime,之后会在vTaskDelayUntil()中自动更新。 */
    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Periodic task is running\r\n" );

        /* The task should execute every 10 milliseconds exactly. */
        vTaskDelayUntil( &xLastWakeTime, ( 10 / portTICK_RATE_MS ) );
    }
}
```

程序清单 16 例6中周期任务的实现代码

图 11 展示了**例 6** 的运行输出结果，图 12 是对看到的行为方式对应的执行流程的解释。

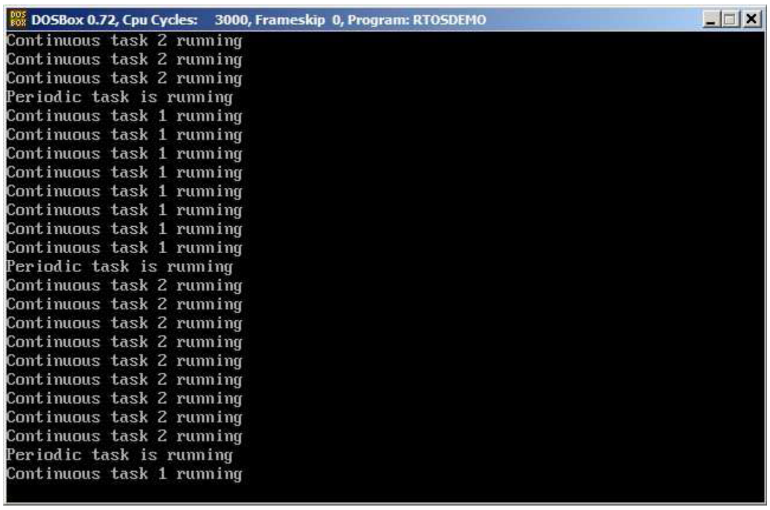


图 11 例 6 的执行输出结果 [1]

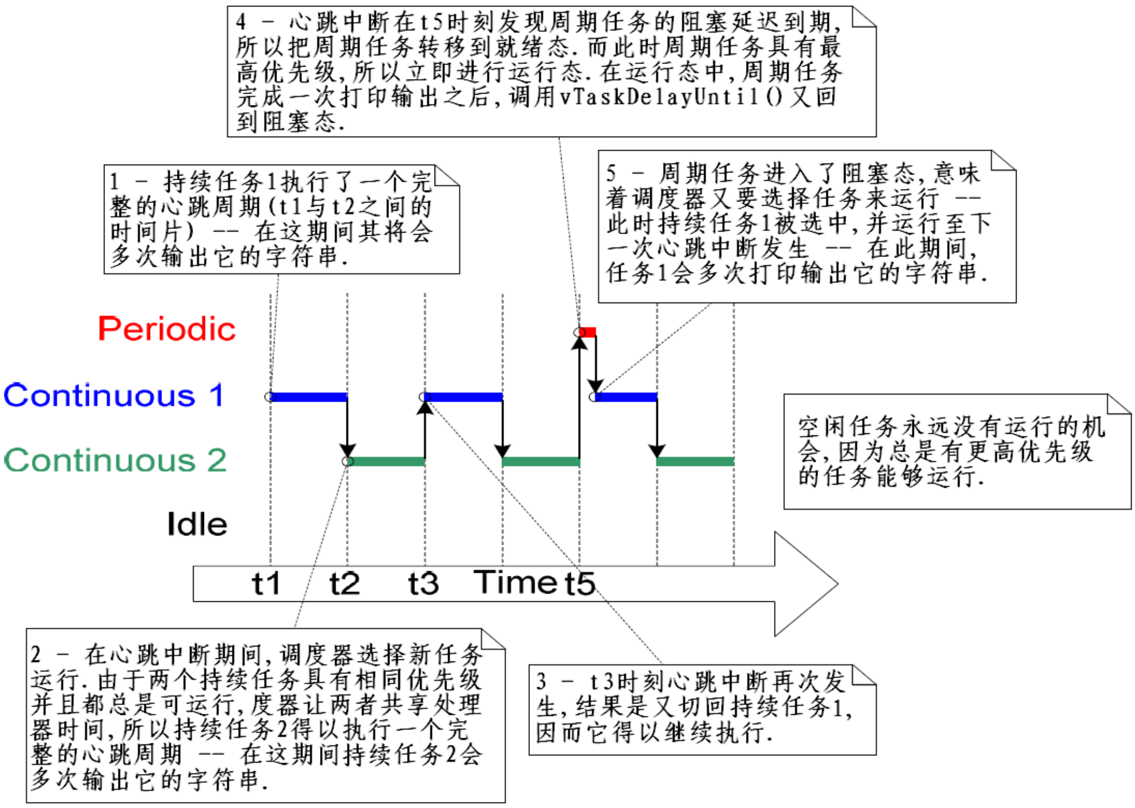


图 12 例 6 的执行流程

[1] 此输出使用 DOSBox 模拟器降低了执行速度，以便在单幅屏幕上观察到完整的系统行为。

1.7 空闲任务与空闲任务钩子函数

例 4 中创建的任务大部份时间都处于阻塞态。这种状态下所有的任务都不可运行，所以也不能被调度器选中。

但处理器总是需要代码来执行——所以至少要有一个任务处于运行态。为了保证这一点，当调用 vTaskStartScheduler()时，调度器会自动创建一个空闲任务。空闲任务是一个非常短小的循环——和最早的示例任务十分相似，总是可以运行。

空闲任务拥有最低优先级(优先级 0)以保证其不会妨碍具有更高优先级的应用任务进入运行态——当然，没有任何限制说是不能把应用任务创建在与空闲任务相同的优先级上；如果需要的话，你一样可以和空闲任务一起共享优先级。

运行在最低优先级可以保证一旦有更高优先级的任务进入就绪态，空闲任务就会立即切出运行态。这一点可以在图 9 的 tn 时刻看出来，当任务 2 退出阻塞态时，空闲任务立即切换出来以让任务 2 执行。任务 2 被看作是抢占(pre-empted)了空闲任务。抢占是自动发生的，也并不需要通知被抢占任务。

空闲任务钩子函数

通过空闲任务钩子函数(或称回调，hook, or call-back)，可以直接在空闲任务中添加应用程序相关的功能。空闲任务钩子函数会被空闲任务每循环一次就自动调用一次。

通常空闲任务钩子函数被用于：

- 执行低优先级，后台或需要不停处理的功能代码。
- 测试处系统处理裕量(空闲任务只会在所有其它任务都不运行时才有机会执行，所以测量出空闲任务占用的处理时间就可以清楚的知道系统有多少富余的处理时间)。
- 将处理器配置到低功耗模式——提供一种自动省电方法，使得在没有任何应用功能需要处理的时候，系统自动进入省电模式。

空闲任务钩子函数的实现限制

空闲任务钩子函数必须遵从以下规则

1. 绝不能阻或挂起。空闲任务只会在其它任务都不运行时才会被执行(除非有应用任务共享空闲任务优先级)。以任何方式阻塞空闲任务都可能导致没有任务能够进入运行态！

2. 如果应用程序用到了 vTaskDelete() AP 函数，则空闲钩子函数必须能够尽快返回。因为在任务被删除后，空闲任务负责回收内核资源。如果空闲任务一直运行在钩子函数中，则无法进行回收工作。

空闲任务钩子函数必须具有程序清单 17 所示的函数名和函数原型。

```
void vApplicationIdleHook( void );
```

程序清单 17 空闲任务钩子函数原型

例 7. 定义一个空闲任务钩子函数

例 4 调用了带阻塞性质的 vTaskDelay() API 函数，会产生大量的空闲时间——在这期间空闲任务会得到执行，因为两个应用任务均处于阻塞态。本例通过空闲钩子函数来使用这些空间时间。具体源代码参见**程序清单 18**。

```
/* Declare a variable that will be incremented by the hook function. */
unsigned long ulIdleCycleCount = 0UL;

/* 空闲钩子函数必须命名为vApplicationIdleHook(),无参数也无返回值。 */
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
}
```

程序清单 18 一个非常简单的空闲钩子函数

FreeRTOSConfig.h 中的配置常量 configUSE_IDLE_HOOK 必须定义为 1，这样空闲任务钩子函数才会被调用。

对应用任务实现函数进行了少量的修改，用以打印输出变量 ulIdleCycleCount 的值，如**程序清单 19** 所示。

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;

    /* The string to print out is passed in via the parameter. Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* 打印输出任务名，以及调用计数器ulIdleCycleCount的值。 */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* Delay for a period for 250 milliseconds. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

程序清单 **19**　示例任务现在用于打印输出 **ulIdleCycleCount** 的值

例 **7** 的输出结果参见图 **13**。从图中可以看出(在我的电脑上)，空闲任务钩子函数在应用任务的每次循环过程中被调用了(非常)接近 4.5 million 次。

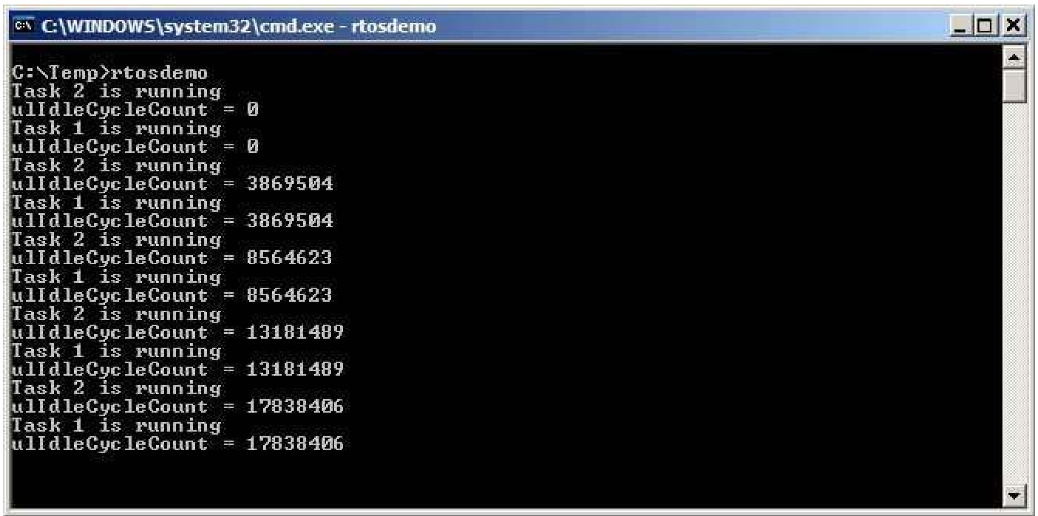


图 **13**　例 **7** 的运行输出结果

1.8 改变任务优先级

vTaskPrioritySet() API 函数

API 函数 vTaskPriofitySet()可以用于在调度器启动后改变任何任务的优先级。

```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
```

程序清单 20 vTaskPrioritySet() API 函数原型

表 4 vTaskPrioritySet() 参数

参数名	描述
pxTask	被修改优先级的任务句柄(即目标任务)——参考 xTaskCreate() API 函数的参数 pxCreatedTask 以了解如何得到任务句柄方面的信息。 任务可以通过传入 NULL 值来修改自己的优先级。
uxNewPriority	目标任务将被设置到哪个优先级上。如果设置的值超过了最大可用优先级(configMAX_PRIORITIES – 1)，则会被自动封顶为最大值。常量 configMAX_PRIORITIES 是在 FreeRTOSConfig.h 头文件中设置的一个编译时选项。

uxTaskPriorityGet() API 函数

uxTaskPriorityGet() API 函数用于查询一个任务的优先级。

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

程序清单 21 uxTaskPriorityGet() API 函数原型

表 5 uxTaskPriorityGet()参数及返回值

参数名	描述
pxTask	被查询任务的句柄(目标任务) ——参考 xTaskCreate() API 函数的参数 pxCreatedTask 以了解如何得到任务句柄方面的信息。 任务可以通过传入 NULL 值来查询自己的优先级。
返回值	被查询任务的当前优先级。

例 8. 改变任务优先级

调度器总是在所有就绪态任务中选择具有最高优先级的任务，并使其进入运行态。本例即是通过调用 vTaskPrioritySet() API 函数来改变两个任务的相对优先级，以达到对调度器这一行为的演示。

在不同的优先级上创建两个任务。这两个任务都没有调用任何会令其进入阻塞态的 API 函数，所以这两个任务要么处于就绪态，要么处于运行态——这种情形下，调度器选择具有最高优先级的任务来执行。

例 8 具有以下行为方式：

- 任务 1(**程序清单 22**)创建在最高优先级，以保证其可以最先运行。任务 1 首先打印输出两个字符串，然后将任务 2(**程序清单 23**)的优先级提升到自己之上。
- 任务 2 一旦拥有最高优先级便启动执行(进入运行态)。由于任何时候只可能有一个任务处于运行态，所以当任务 2 运行时，任务 1 处于就绪态。
- 任务 2 打印输出一个信息，然后把自己的优先级设回低于任务 1 的初始值。
- 任务 2 降低自己的优先级意味着任务 1 又成为具有最高优先级的任务，所以任务 1 重新进入运行态，任务 2 被强制切入就绪态。

```
void vTask1( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;
    /* 本任务将会比任务2更先运行，因为本任务创建在更高的优先级上。任务1和任务2都不会阻塞，所以两者要
    么处于就绪态，要么处于运行态。
    查询本任务当前运行的优先级 – 传递一个NULL值表示说“返回我自己的优先级”。 */
    uxPriority = uxTaskPriorityGet( NULL );
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task1 is running\r\n" );

        /* 把任务2的优先级设置到高于任务1的优先级，会使得任务2立即得到执行(因为任务2现在是所有任务
        中具有最高优先级的任务)。注意调用vTaskPrioritySet()时用到的任务2的句柄。程序清单24将展示
        如何得到这个句柄。 */
        vPrintString( "About to raise the Task2 priority\r\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* 本任务只会在其优先级高于任务2时才会得到执行。因此，当此任务运行到这里时，任务2必然已经执
        行过了，并且将其自身的优先级设置回比任务1更低的优先级。 */
    }
}
```

程序清单 **22**　例 **8** 中任务 **1** 的实现代码

```
void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;
    /* 任务1比本任务更先启动，因为任务1创建在更高的优先级。任务1和任务2都不会阻塞，所以两者要么处于
    就绪态，要么处于运行态。
    查询本任务当前运行的优先级 – 传递一个NULL值表示说“返回我自己的优先级”。 */
    uxPriority = uxTaskPriorityGet( NULL );
    for( ;; )
    {
        /* 当任务运行到这里，任务1必然已经运行过了，并将本身务的优先级设置到高于任务1本身。 */
        vPrintString( "Task2 is running\r\n" );

        /* 将自己的优先级设置回原来的值。传递NULL句柄值意味“改变我己自的优先级”。把优先级设置到低
        于任务1使得任务1立即得到执行 – 任务1抢占本任务。 */
        vPrintString( "About to lower the Task2 priority\r\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

程序清单 **23**　例 **8** 中的任务 **2** 实现代码

任务在查询和修改自己的优先级时，并没有使用一个有效的句柄——使用 NULL 代替。只有在某个任务需要引用其它任务的时候才会用到任务句柄。好比任务 1 想要改变任务 2 的优先级，为了让任务 1 能够使用到任务 2 的句柄，在任务 2 被创建时其句柄就被获得并保存下来，就像**程序清单 24** 注释中重点提示的一样。

```
/* 声明变量用于保存任务2的句柄。 */
xTaskHandle xTask2Handle;

int main( void )
{
    /* 任务1创建在优先级2上。任务参数没有用到，设为NULL。任务句柄也不会用到，也设为NULL */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );
    /* The task is created at priority 2 ______^. */

    /* 任务2创建在优先级1上 - 此优先级低于任务1。任务参数没有用到，设为NULL。但任务2的任务句柄会被
    用到，故将xTask2Handle的地址传入。 */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );
    /* The task handle is the last parameter _____^^^^^^^^^^^^^ */

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    CHAPTER 5 provides more information on memory management. */
    for( ;; );
}
```

程序清单 24　例 8 中 main()函数实现代码

图 14 展示了**例 8** 的执行流程，**例 8** 的运行输出结果参见**图 15**。

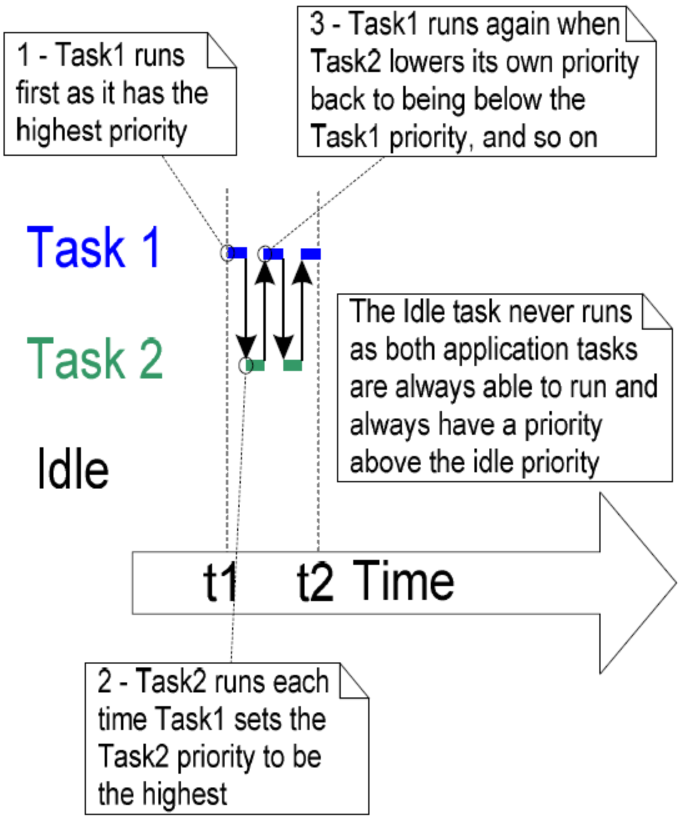


图 **14**　例 **8** 的执行流程

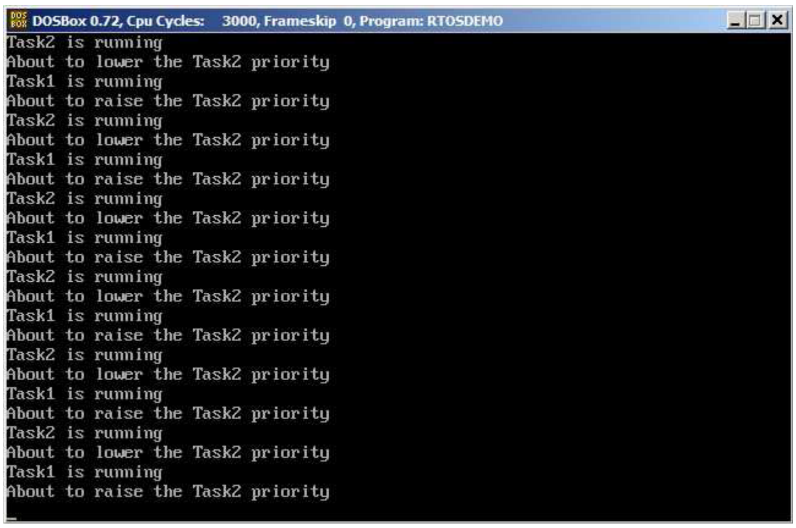


图 **15**　例 **8** 的运行输出结果

1.9 删除任务

vTaskDelete() API 函数

任务可以使用 API 函数 vTaskDelete()删除自己或其它任务。

任务被删除后就不复存在，也不会再进入运行态。

空闲任务的责任是要将分配给已删除任务的内存释放掉。因此有一点很重要，那就是使用 vTaskDelete() API 函数的任务千万不能把空闲任务的执行时间饿死。

需要说明一点，只有内核为任务分配的内存空间才会在任务被删除后自动回收。任务自己占用的内存或资源需要由应用程序自己显式地释放。

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

程序清单 25　vTaskDelete() API 函数原型

表 6　vTaskDelete()参数

参数名	描述
pxTaskToDelete	被删除任务的句柄(目标任务) —— 参考 xTaskCreate() API 函数的参数 pxCreatedTask 以了解如何得到任务句柄方面的信息。 任务可以通过传入 NULL 值来删除自己。

例 9. 删除任务

这是一个非常简单的范例，其行为如下：

- 任务 1 则 main()创建在优先级 1 上。任务 1 运行时，以优先级 2 创建任务 2。现在任务 2 具有最高优先级，所以会立即得到执行。main()函数的源代码参见程序清单 26，任务 1 的实现代码参见程序清单 27。
- 任务 2 什么也没有做，只是删除自己。可以通过传递 NULL 值以 vTaskDelete()来删除自己，但是为了纯粹的演示，传递的是任务自己的句柄。任务 2 的实现源代码见程序清单 28。

- 当任务 2 被自己删除之后，任务 1 成为最高优先级的任务，所以继续执行，调用 vTaskDelay()阻塞一小段时间。
- 当任务 1 进入阻塞状态后，空闲任务得到执行的机会。空闲任务会释放内核为已删除的任务 2 分配的内存。
- 任务 1 离开阻塞态后，再一次成为就绪态中具有最高优先级的任务，因此会抢占空闲任务。又再一次创建任务 2，如此往复。

```
int main( void )
{
    /* 任务1创建在优先级1上 */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );
    /* The task is created at priority 1 ______^. */

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* main() should never reach here as the scheduler has been started. */
    for( ;; );
}
```

程序清单 26　例 9 中的 main()函数实现

```
void vTask1( void *pvParameters )
{
const portTickType xDelay100ms = 100 / portTICK_RATE_MS;
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task1 is running\r\n" );

        /* 创建任务2为最高优先级。 */
        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter _____^^^^^^^^^^^^^ */

        /* 因为任务2具有最高优先级，所以任务1运行到这里时，任务2已经完成执行，删除了自己。任务1得以
        执行，延迟100ms */
        vTaskDelay( xDelay100ms );
    }
}
```

程序清单 27　例 9 中任务 1 的实现代码

```
void vTask2( void *pvParameters )
{
    /* 任务2什么也没做，只是删除自己。删除自己可以传入NULL值，这里为了演示，还是传入其自己的句柄。 */
    vPrintString( "Task2 is running and about to delete itself\r\n" );
    vTaskDelete( xTask2Handle );
}
```

程序清单 **28**　例 **9** 中的任务 **2** 实现代码

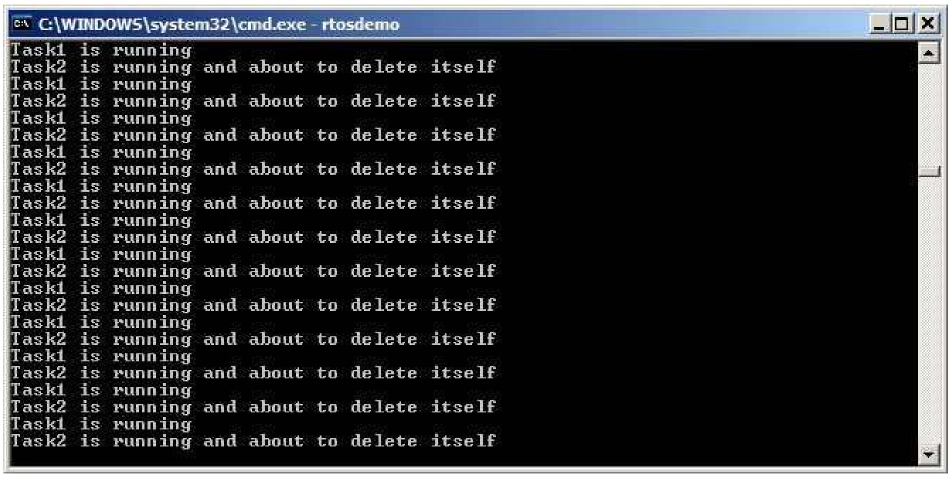


图 **16**　例 **9** 的运行输出结果

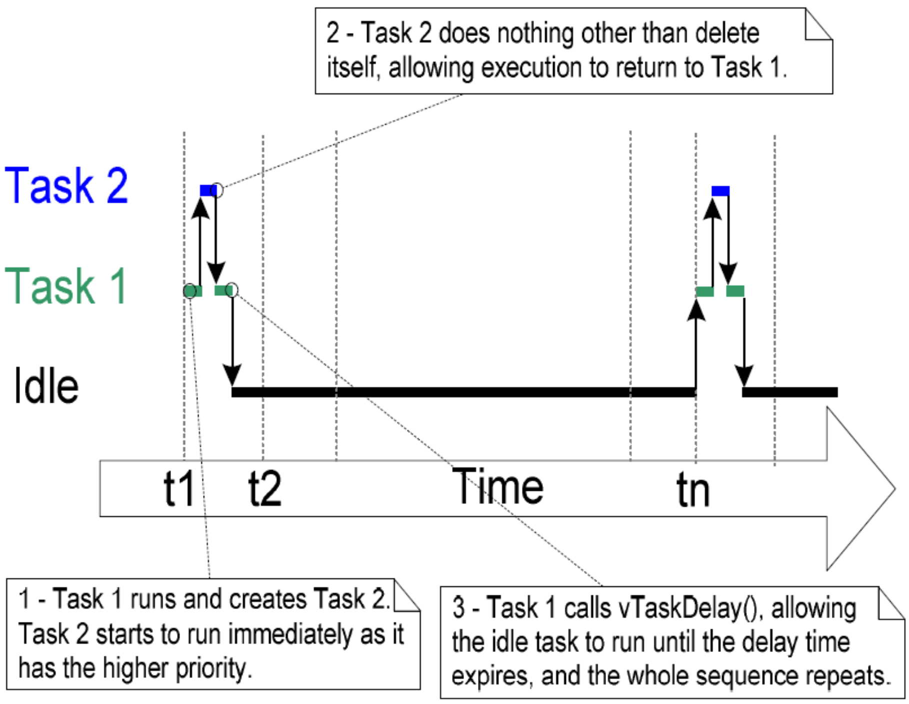


图 **17**　例 **9** 执行流程

1.10 调度算法 – 简述

优先级抢占式调度

本章的示例程序已经演示了 FreeRTOS 在什么时候以及以什么方式选择一个什么样的任务来执行。

- 每个任务都赋予了一个优先级。
- 每个任务都可以存在于一个或多个状态。
- 在任何时候都只有一个任务可以处于运行状态。
- 调度器总是在所有处于就绪态的任务中选择具有最高优先级的任务来执行。

这种类型的调度方案被称为"固定优先级抢占式调度"。所谓"固定优先级"是指每个任务都被赋予了一个优先级，这个优先级不能被内核本身改变(只能被任务修改)。"抢占式"是指当任务进入就绪态或是优先级被改变时，如果处于运行态的任务优先级更低，则该任务总是抢占当前运行的任务。

任务可以在阻塞状态等待一个事件，当事件发生时其将自动回到就绪态。时间事件发生在某个特定的时刻，比如阻塞超时。时间事件通常用于周期性或超时行为。任务或中断服务例程往队列发送消息或发送任务一种信号量，都将触发同步事件。同步事件通常用于触发同步行为，比如某个外围的数据到达了。

图 18 通过图示某个应用程序的执行流程展现了抢占式调度的行为方式。

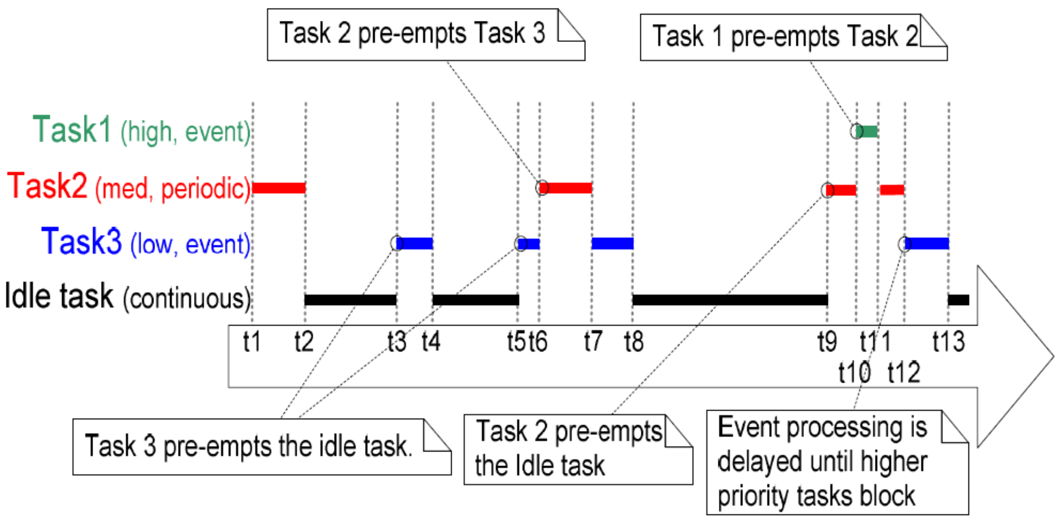


图 18 执行流程中的主要抢占点

如图 **18** 中所示：

1. 空闲任务

   空闲任务具有最低优先级，所以每当有更高优先级任务处于就绪态是，空闲任务就会被抢占 —— 如图中 t3, t5 和 t9 时刻。

2. 任务 3

   任务 3 是一个事件驱动任务。其工作在一个相对较低的优先级，但优先级高于空闲任务。其大部份时间都在阻塞态等待其关心的事件。每当事件发生时其就从阻塞态转移到就绪态。FreeRTOS 中所有的任务间通信机制(队列，信号量等)都可以通过这种方式用于发送事件以及让任务解除阻塞。

   事件在 t3，t5 以及 t9 至 t12 之间的某个时刻发生。发生在 t3 和 t5 时刻的事件可以立即被处理，因为这些时刻任务 3 在所有可运行任务中优先级最高。发生在 t9 至 t12 之间某个时刻的事件不会得到立即处理，需要一直等到 t12 时刻。因为具有更高优先级的任务 1 和任务 2 尚在运行中，只有到了 t12 时刻，这两个任务进入阻塞态，使得任务 3 成为具有最高优先级的就绪态任务。

3. 任务 2

   任务 2 是一个周期性任务，其优先级高于任务 3 并低于任务 1。根据周期间隔，任务 2 期望在 t1，t6 和 t9 时刻执行。

   在 t6 时刻任务 3 处于运行态，但是任务 2 相对具有更高的优先级，所以会抢占任务 3，并立即得到执行。任务 2 完成处理后，在 t7 时刻返回阻塞态。同时，任务 3 得以重新进入运行态，继续完成处理。任务 3 在 t8 时刻进入阻塞状态。

4. 任务 1

   任务 1 也是一个事件驱动任务。任务 1 在所有任务中具有最高优先级，因此可以抢占系统中的任何其它任务。在图中看到，任务 1 的事件只是发生在在 t10 时刻，此时任务 1 抢占了任务 2。只有当任务 1 在 t11 时刻再次进入阻塞态之后，任务 2 才得以机会继续完成处理。

选择任务优先级

从图 **18** 中可以看到优先级分配是如何从根本上影响应用程序行为的。

作为一种通用规则，完成硬实时功能的任务优先级会高于完成软件时功能任务的优先级。但其它一些因素，比如执行时间和处理器利用率，都必须纳入考虑范围，以保证应用程序不会超过硬实时的需求限制。

单调速率调度(Rate Monotonic Scheduling, RMS)是一种常用的优先级分配技术。其根据任务周期性执行的速率来分配一个唯一的优先级。具有最高周期执行频率的任务赋予高最优先级；具有最低周期执行频率的任务赋予最低优先级。这种优先级分配方式被证明了可以最大化整个应用程序的可调度性(schedulability)，但是运行时间不定以及并非所有任务都具有周期性，会使得对这种方式的全面计算变得相当复杂。

协作式调度

本书专注于抢占式调度。FreeRTOS 可以选择采用协作式调度。

采用一个纯粹的协作式调度器，只可能在运行态任务进入阻塞态或是运行态任务显式调用 taskYIELD()时，才会进行上下文切换。任务永远不会被抢占，而具有相同优先级的任务也不会自动共享处理器时间。协作式调度的这作工作方式虽然比较简单，但可能会导致系统响应不够快。

实现混合调度方案也是可行的，这需要在中断服务例程中显式地进行上下文切换，从而允许同步事件产生抢占行为，但时间事件却不行。这样做的结果是得到了一个没有时间片机制的抢占式系统。或许这正是所期望的，因为获得了效率，并且这也是一种常用的调度器配置。

第二章

队列管理

2.1 概览

基于 FreeRTOS 的应用程序由一组独立的任务构成——每个任务都是具有独立权限的小程序。这些独立的任务之间很可能会通过相互通信以提供有用的系统功能。FreeRTOS 中所有的通信与同步机制都是基于队列实现的。

本章期望让读者了解以下事情

- 如何创建一个队列
- 队列如何管理其数据
- 如何向队列发送数据
- 如何从队列接收数据
- 队列阻塞是什么意思
- 往队列发送和从队列接收时，任务优先级会有什么样的影响

本章仅涵盖任务之间的通信。任务与中断之间的通信将在第三章讲述。

2.2 队列的特性

数据存储

队列可以保存有限个具有确定长度的数据单元。队列可以保存的最大单元数目被称为队列的“深度”。在队列创建时需要设定其深度和每个单元的大小。

通常情况下，队列被作为 FIFO(先进先出)使用，即数据由队列尾写入，从队列首读出。当然，由队列首写入也是可能的。

往队列写入数据是通过字节拷贝把数据复制存储到队列中；从队列读出数据使得把队列中的数据拷贝删除。**图 19** 展现了队列的写入与读出过程，以及读写操作对队列中数据的影响。

可被多任务存取

队列是具有自己独立权限的内核对象，并不属于或赋予任何任务。所有任务都可以向同一队列写入和读出。一个队列由多方写入是经常的事，但由多方读出倒是很少遇到。

读队列时阻塞

当某个任务试图读一个队列时，其可以指定一个阻塞超时时间。在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列数据有效。当其它任务或中断服务例程往其等待的队列中写入了数据，该任务将自动由阻塞态转移为就绪态。当等待的时间超过了指定的阻塞时间，即使队列中尚无有效数据，任务也会自动从阻塞态转移为就绪态。

由于队列可以被多个任务读取，所以对单个队列而言，也可能有多个任务处于阻塞状态以等待队列数据有效。这种情况下，一旦队列数据有效，只会有一个任务会被解除阻塞，这个任务就是所有等待任务中优先级最高的任务。而如果所有等待任务的优先级相同，那么被解除阻塞的任务将是等待最久的任务。

写队列时阻塞

同读队列一样，任务也可以在写队列时指定一个阻塞超时时间。这个时间是当被写队列已满时，任务进入阻塞态以等待队列空间有效的最长时间。

由于队列可以被多个任务写入，所以对单个队列而言，也可能有多个任务处于阻塞状态以等待队列空间有效。这种情况下，一旦队列空间有效，只会有一个任务会被解除阻塞，这个任务就是所有等待任务中优先级最高的任务。而如果所有等待任务的优先级相同，那么被解除阻塞的任务将是等待最久的任务。

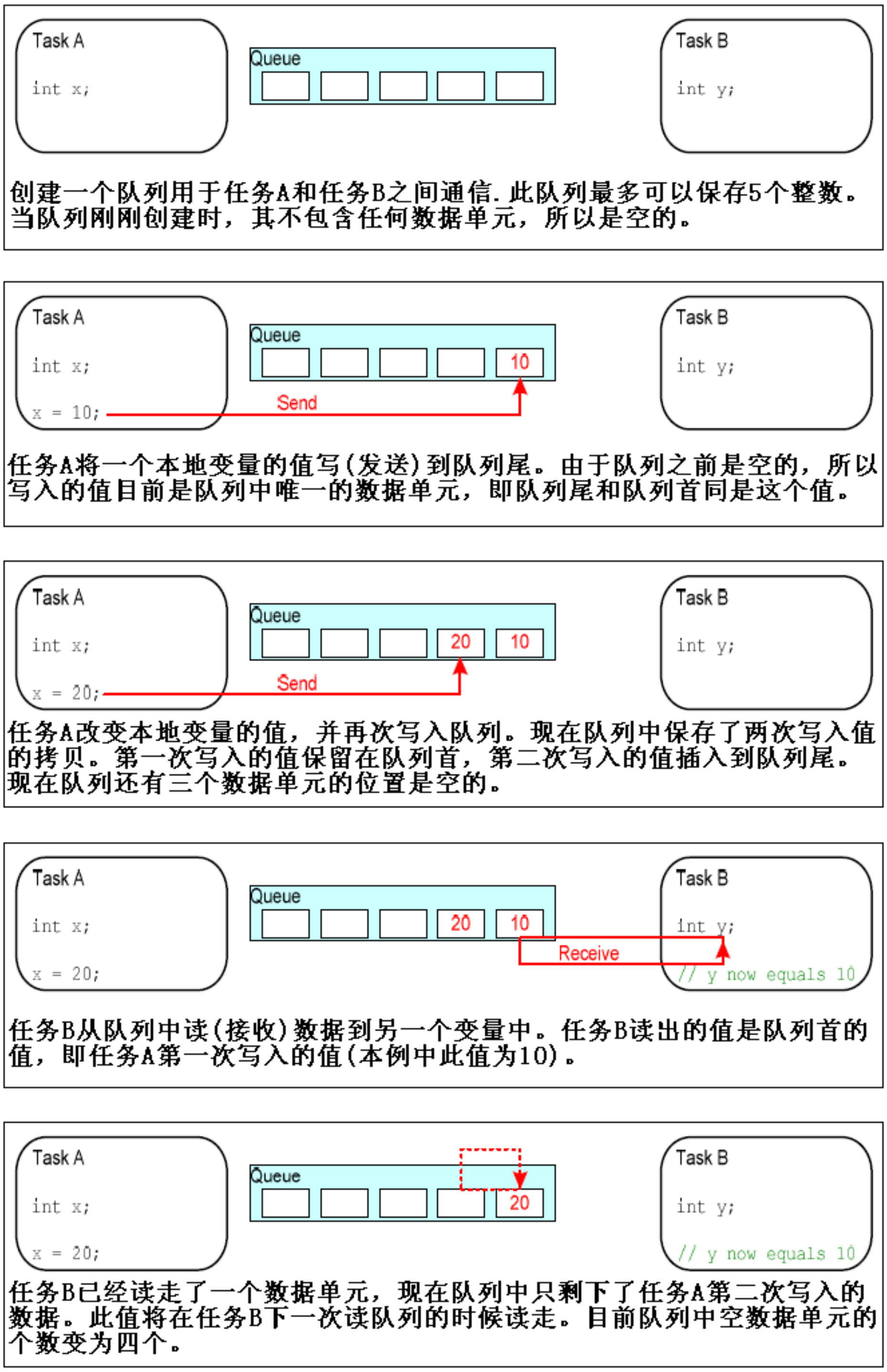


图 **19** 队列读写过程示例

2.3 使用队列

xQueueCreate() API 函数

队列在使用前必须先被创建。

队列由声明为 xQueueHandle 的变量进行引用。xQueueCreate()用于创建一个队列，并返回一个 xQueueHandle 句柄以便于对其创建的队列进行引用。

当创建队列时，FreeRTOS 从堆空间中分配内存空间。分配的空间用于存储队列数据结构本身以及队列中包含的数据单元。如果内存堆中没有足够的空间来创建队列，xQueueCreate()将返回 NULL。第五章会有关于内存堆管理的更多信息。

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize );
```

程序清单 **29 xQueueCreate() API** 函数原型

表 7 xQueueCreate()参数与返回值

参数名	描述
uxQueueLength	队列能够存储的最大单元数目，即队列深度。
uxItemSize	队列中数据单元的长度，以字节为单位。
返回值	NULL 表示没有足够的堆空间分配给队列而导致创建失败。 非 NULL 值表示队列创建成功。此返回值应当保存下来，以作为操作此队列的句柄。

xQueueSendToBack() 与 xQueueSendToFront() API 函数

如同函数名字面意思所期望的一样，xQueueSendToBack()用于将数据发送到队列尾；而 xQueueSendToFront()用于将数据发送到队列首。

xQueueSend()完全等同于 xQueueSendToBack()。

但切记不要在中断服务例程中调用 xQueueSendToFront() 或 xQueueSendToBack()。系统提供中断安全版本的 xQueueSendToFrontFromISR()与 xQueueSendToBackFromISR()用于在中断服务中实现相同的功能。这些将在第三章中详述。

```
portBASE_TYPE xQueueSendToFront(   xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait );
```

程序清单 30 The xQueueSendToFront() API函数原型

```
portBASE_TYPE xQueueSendToBack(    xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait );
```

程序清单 31 The xQueueSendToBack() API 函数原型

表 8 xQueueSendToFront()与 xQueueSendToBack()函数参数及返回值

参数名	描述
xQueue	目标队列的句柄。这个句柄即是调用 xQueueCreate()创建该队列时的返回值。
pvItemToQueue	发送数据的指针。其指向将要复制到目标队列中的数据单元。 由于在创建队列时设置了队列中数据单元的长度，所以会从该指针指向的空间复制对应长度的数据到队列的存储区域。
xTicksToWait	阻塞超时时间。如果在发送时队列已满，这个时间即是任务处于阻塞态等待队列空间有效的最长等待时间。

如果 xTicksToWait 设为 0，并且队列已满，则 xQueueSendToFront()与 xQueueSendToBack()均会立即返回。

阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位。

如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制。

返回值

有两个可能的返回值:

1. pdPASS

返回 pdPASS 只会有一种情况，那就是数据被成功发送到队列中。

如果设定了阻塞超时时间(xTicksToWait 非 0)，在函数返回之前任务将被转移到阻塞态以等待队列空间有效—在超时到来前能够将数据成功写入到队列，函数则会返回 pdPASS。

2. errQUEUE_FULL

如果由于队列已满而无法将数据写入，则将返回 errQUEUE_FULL。

如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列空间有效。但直到超时也没有其它任务或是中断服务例程读取队列而腾出空间，函数则会返回 errQUEUE_FULL。

xQueueReceive()与 xQueuePeek() API 函数

xQueueReceive()用于从队列中接收(读取）数据单元。接收到的单元同时会从队列中删除。

xQueuePeek()也是从从队列中接收数据单元，不同的是并不从队列中删出接收到的单元。xQueuePeek()从队列首接收到数据后，不会修改队列中的数据，也不会改变数据在队列中的存储序顺。

切记不要在中断服务例程中调用 xQueueRceive()和 xQueuePeek()。中断安全版本的替代 API 函数 xQueueReceiveFromISR()将会在第三章中讲述。

```
portBASE_TYPE xQueueReceive(   xQueueHandle xQueue,
                               const void * pvBuffer,
                               portTickType xTicksToWait );
```

图 20 xQueueReceive() API 函数原型

（译者注-怎么成“图20”呢？这是原文中的Bug，好在并不影响什么）

```
portBASE_TYPE xQueuePeek(  xQueueHandle xQueue,
                           const void * pvBuffer,
                           portTickType xTicksToWait );
```

程序清单 32 xQueuePeek() API 函数原型

表 9 xQueueReceive()与 xQueuePeek()函数参数与返回值

参数名	描述
xQueue	被读队列的句柄。这个句柄即是调用 xQueueCreate()创建该队列时的返回值。
pvBuffer	接收缓存指针。其指向一段内存区域，用于接收从队列中拷贝来的数据。 数据单元的长度在创建队列时就已经被设定，所以该指针指向的内存区域大小应当足够保存一个数据单元。
xTicksToWait	阻塞超时时间。如果在接收时队列为空，则这个时间是任务处于

阻塞状态以等待队列数据有效的最长等待时间。

如果 xTicksToWait 设为 0，并且队列为空，则 xQueueRecieve() 与 xQueuePeek()均会立即返回。

阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位。

如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制。

返回值

有两个可能的返回值:

1. pdPASS

只有一种情况会返回 pdPASS，那就是成功地从队列中读到数据。

如果设定了阻塞超时时间(xTicksToWait 非 0)，在函数返回之前任务将被转移到阻塞态以等待队列数据有效—在超时到来前能够从队列中成功读取数据，函数则会返回 pdPASS。

2. errQUEUE_FULL

如果在读取时由于队列已空而没有读到任何数据，则将返回 errQUEUE_FULL。

如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列数据有效。但直到超时也没有其它任务或是中断服务例程往队列中写入数据，函数则会返回 errQUEUE_FULL。

uxQueueMessagesWaiting() API 函数

uxQueueMessagesWaiting()用于查询队列中当前有效数据单元个数。

切记不要在中断服务例程中调用 uxQueueMessagesWaiting()。应当在中断服务中使用其中断安全版本 uxQueueMessagesWaitingFromISR()。

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

程序清单 33 uxQueueMessagesWaiting() API 函数原型

表 10 uxQueueMessagesWaiting()函数参数及返回值

参数名	描述
xQueue	被查询队列的句柄。这个句柄即是调用 xQueueCreate()创建该队列时的返回值。
返回值	当前队列中保存的数据单元个数。返回 0 表明队列为空。

例 10. 读队列时阻塞

本例示范创建一个队列，由多个任务往队列中写数据，以及从队列中把数据读出。这个队列创建出来保存 long 型数据单元。往队列中写数据的任务没有设定阻塞超时时间，而读队列的任务设定了超时时间。

往队列中写数据的任务的优先级低于读队列任务的优先级。这意味着队列中永远不会保持超过一个的数据单元。因为一旦有数据被写入队列，读队列任务立即解除阻塞，抢占写队列任务，并从队列中接收数据，同时数据从队列中删除—队列再一次变为空队列。

程序清单 34 展现了写队列任务的代码实现。这个任务被创建了两个实例，一个不停地往队列中写数值 100，而另一个实例不停地往队列中写入数值 200。任务的入口参数被用来为每个实例传递各自的写入值。

```
static void vSenderTask( void *pvParameters )
{
long lValueToSend;
portBASE_TYPE xStatus;

    /* 该任务会被创建两个实例，所以写入队列的值通过任务入口参数传递 – 这种方式使得每个实例使用不同的
    值。队列创建时指定其数据单元为long型，所以把入口参数强制转换为数据单元要求的类型 */
    lValueToSend = ( long ) pvParameters;

    /* 和大多数任务一样，本任务也处于一个死循环中 */
    for( ;; )
    {
        /* 往队列发送数据
        第一个参数是要写入的队列。队列在调度器启动之前就被创建了，所以先于此任务执行。

        第二个参数是被发送数据的地址，本例中即变量lValueToSend的地址。

        第三个参数是阻塞超时时间 – 当队列满时，任务转入阻塞状态以等待队列空间有效。本例中没有设定超
        时时间，因为此队列决不会保持有超过一个数据单元的机会，所以也决不会满。
         */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
        if( xStatus != pdPASS )
        {
            /* 发送操作由于队列满而无法完成 – 这必然存在错误，因为本例中的队列不可能满。 */
            vPrintString( "Could not send to the queue.\r\n" );
        }

        /* 允许其它发送任务执行。 taskYIELD()通知调度器现在就切换到其它任务，而不必等到本任务的时
        间片耗尽 */
        taskYIELD();
    }
}
```

程序清单 **34** 例 **10** 中的写队列任务实现代码

程序清单 35 展现了读队列任务的代码实现。读队列任务设定了 100 毫秒的阻塞超时时间，所以会进入阻塞态以等待队列数据有效。一旦队列中数据单元有效，或者即使队列数据无效但等待时间超过 100 毫秒，此任务将会解除阻塞。在本例中，将永远不会出现 100 毫秒超时，因为有两个任务在不停地往队列中写数据。

```
static void vReceiverTask( void *pvParameters )
{
/* 声明变量，用于保存从队列中接收到的数据。 */
long lReceivedValue;
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* 本任务依然处于死循环中。 */
    for( ;; )
    {
        /* 此调用会发现队列一直为空，因为本任务将立即删除刚写入队列的数据单元。 */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }

        /* 从队列中接收数据
        第一个参数是被读取的队列。队列在调度器启动之前就被创建了，所以先于此任务执行。

        第二个参数是保存接收到的数据的缓冲区地址，本例中即变量lReceivedValue的地址。此变量类型与
        队列数据单元类型相同，所以有足够的大小来存储接收到的数据。

        第三个参数是阻塞超时时间 – 当队列空时，任务转入阻塞状态以等待队列数据有效。本例中常量
        portTICK_RATE_MS用来将100毫秒绝对时间转换为以系统心跳为单位的时间值。
         */
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
        if( xStatus == pdPASS )
        {
            /* 成功读出数据，打印出来。 */
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else
        {
            /* 等待100ms也没有收到任何数据。
            必然存在错误，因为发送任务在不停地往队列中写入数据 */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
```

程序清单 **35** 例 **10** 中的读队列任务实现代码

程序清单 36 包含了 main()函数的实现。其在启动调度器之前创建了一个队列和三个任务。尽管对任务的优先级的设计使得队列实际上在任何时候都不可能多于一个数据单元，本例代码还是创建了一个可以保存最多 5 个 long 型值的队列。

```
/* 声明一个类型为 xQueueHandle 的变量. 其用于保存队列句柄，以便三个任务都可以引用此队列 */
xQueueHandle xQueue;
int main( void )
{
    /* 创建的队列用于保存最多5个值，每个数据单元都有足够的空间来存储一个long型变量 */
    xQueue = xQueueCreate( 5, sizeof( long ) );
    if( xQueue != NULL )
    {
        /* 创建两个写队列任务实例，任务入口参数用于传递发送到队列的值。所以一个实例不停地往队列发送
        100，而另一个任务实例不停地往队列发送200。两个任务的优先级都设为1。 */
        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

        /* 创建一个读队列任务实例。其优先级设为2，高于写任务优先级 */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

        /* 启动调度器，任务开始执行 */
        vTaskStartScheduler();
    }
    else
    {
        /* 队列创建失败*/
    }

    /* 如果一切正常，main()函数不应该会执行到这里。但如果执行到这里，很可能是内存堆空间不足导致空闲
    任务无法创建。第五章有讲述更多关于内存管理方面的信息 */
    for( ;; );
}
```

程序清单 36 例 10 中的 main()函数实现代码

写队列任务在每次循环中都调用 taskYIELD()。taskYIELD()通知调度器立即进行任务切换，而不必等到当前任务的时间片耗尽。某个任务调用 taskYIELD()等效于其自愿放弃运行态。由于本例中两个写队列任务具有相同的任务优先级，所以一旦其中一个任务调用了 taskYIELD()，另一个任务将会得到执行 — 调用 taskYIELD()的任务转移到就绪态，同时另一个任务进入运行态。这样就可以使得这两个任务轮翻地往队列发送数

据。从图 **21** 中可以看到例 **10** 的输出结果。

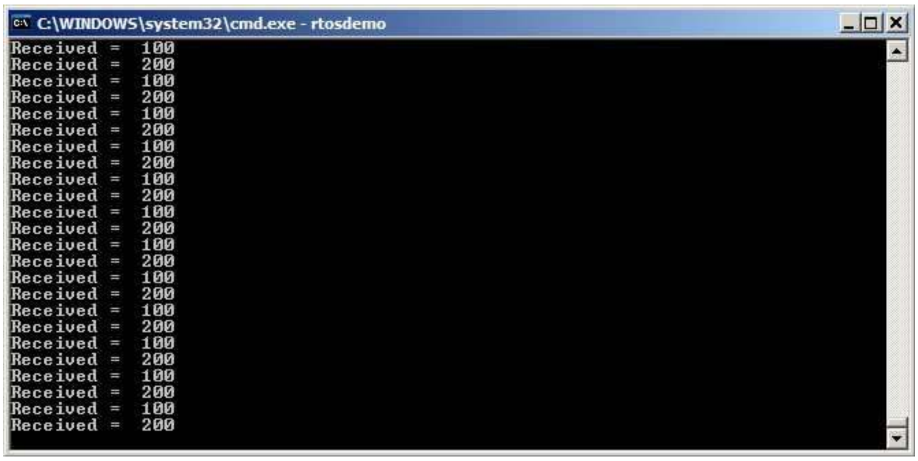


图 **21** 例 **10** 输出结果

图 **22** 展示了本例的执行流程

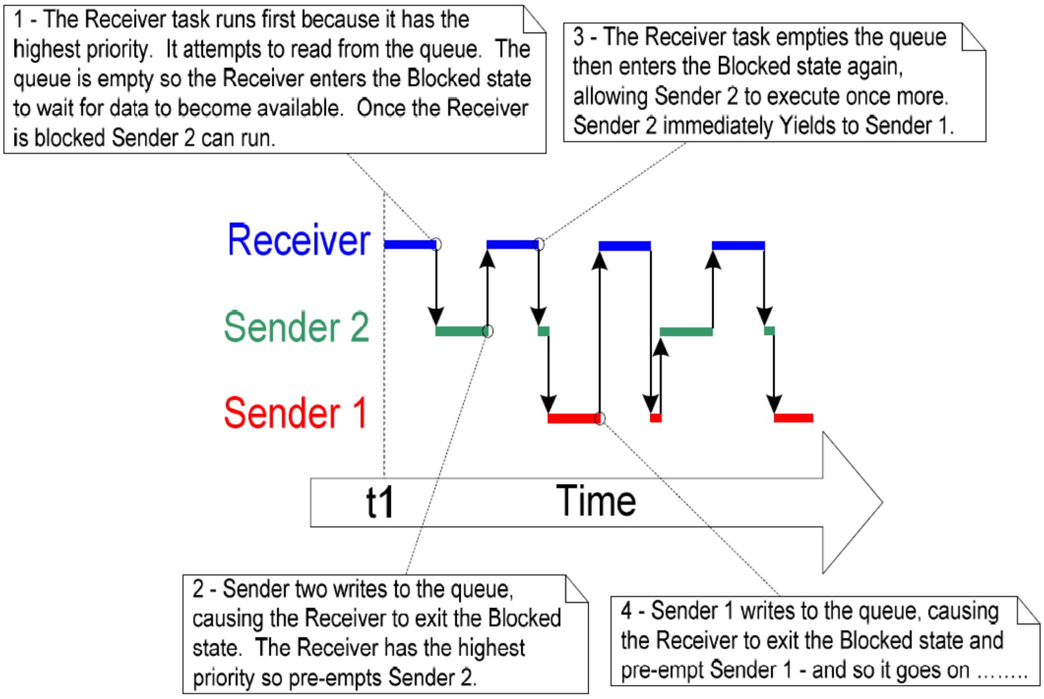


图 **22** 例 **10** 中代码的执行流程

使用队列传递复合数据类型

一个任务从单个队列中接收来自多个发送源的数据是经常的事。通常接收方收到数据后，需要知道数据的来源，并根据数据的来源决定下一步如何处理。一个简单的方式就是利用队列传递结构体，结构体成员中就包含了数据信息和来源信息。**图 23** 对这一方案进行了展现。

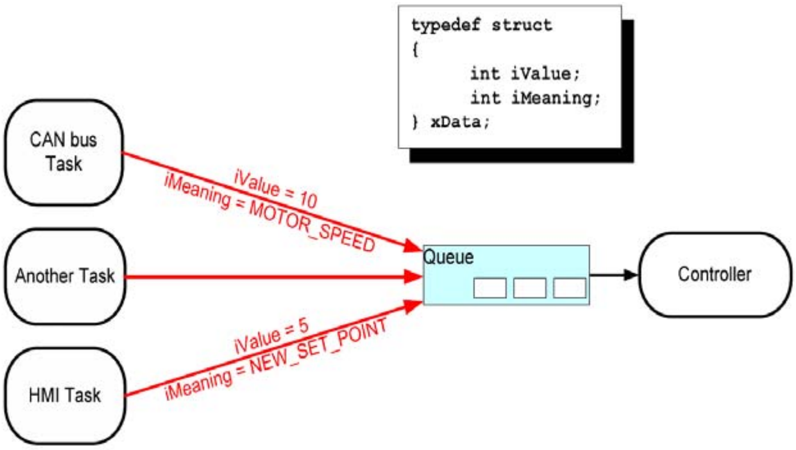


图 **23** 结构体被用于队列传递的一种情形

从图 **23** 中可以看出:

- 创建一个队列用于保存类型为 xData 的结构体数据单元。结构体成员包括了一个数据值和表示数据含义的编码，两者合为一个消息可以一次性发送到队列。
- 中央控制任务用于完成主要的系统功能。其必须对队列中传来的输入和其它系统状态的改变作出响应。
- CAN 总线任务用于封装 CAN 总线的接口功能。当 CAN 总线任务收到并解码一个消息后，其将把解码后的消息放到 xData 结构体中发往控制任务。结构体的 iMeaning 成员用于让中央控制任务知道这个数据是用来干什么的 — 从图中的描述可以看出，这个数据表示电机速度。结构体的 iValue 成员可以让中央控制任务知道电机的实际速度值。
- 人机接口(HMI)任务用于对所有的人机接口功能进行封装。设备操作员可能通过各种方式进行命令输入和参数查询，人机接口任务需要对这些操作进行检测并解析。当接收到一个新的命令后，人机接口任务通过 xData 结构将命令发送到中央控制任务。结构体的 iMeaning 成员用于让中央控制任务知道这个数据是用来干什么的 — 从图中的描述可以看出，这个数据表示一个新的参数设置。结构体的 iValue 成员可以让中央控制任务知道具体的设置值。

例 11. 写队列时阻塞 / 往队列发送结构体

例 11 与**例 10** 类似，只是写队列任务与读队列任务的优先级交换了，即读队列任务的优先级低于写队列任务的优先级。并且本例中的队列用于在任务间传递结构体数据，而非简单的长整型数据。

程序清单 37 展示了**例 11** 中要用到的结构体定义。

```
/* 定义队列传递的结构类型。 */
typedef struct
{
    unsigned char ucValue;
    unsigned char ucSource;
} xData;

/* 声明两个xData类型的变量，通过队列进行传递。 */
static const xData xStructsToSend[ 2 ] =
{
    { 100, mainSENDER_1 }, /* Used by Sender1. */
    { 200, mainSENDER_2 }  /* Used by Sender2. */
};
```

程序清单 37 定义队列传递的数据结构，并声明此类型的两个变量在本例中使用.

在**例 10** 中读队列任务具有最高优先级，所以队列不会拥有一个以上的数据单元。这是因为一旦数据被写队列任务写进队列，读队列任务立即抢占写队列任务，把刚写入的数据单元读走。在**例 11** 中，写队列任务具有最高优先级，所以队列正常情况下一直是处于满状态。这是因为一旦读队列任务从队列中读走一个数据单元，某个写队列任务就会立即抢占读队列任务，把刚刚读走的位置重新写入，之后便又转入阻塞态以等待队列空间有效。

程序清单 38 是写队列任务的实现代码。写队列任务指定了 100 毫秒的阻塞超时时间，以便在队列满时转入阻塞态以等待队列空间有效。进入阻塞态后，一旦队列空间有效，或是等待超过了 100 毫秒队列空间尚无效，其将解除阻塞。在本例中，将永远不会出现 100 毫秒超时的情况，因为读队列任务在不停地从队列中读出数据从而腾出队列数据空间。

```
static void vSenderTask( void *pvParameters )
{
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send to the queue.
        第二个参数是将要发送的数据结构地址。这个地址是从任务入口参数中传入，所以直接使用
        pvParameters.

        第三个参数是阻塞超时时间 – 当队列满时，任务转入阻塞态等待队列空间有效的最长时间。指定超时时
        间是因为写队列任务的优先级高于读任务的优先级。所以队列如预期一样很快写满，写队列任务就会转入
        阻塞态，此时读队列任务才会得以执行，才能从队列中把数据读走。 */
        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );
        if( xStatus != pdPASS )
        {
            /* 写队列任务无法将数据写入队列，直至100毫秒超时。
            这必然存在错误，因为只要写队列任务进入阻塞态，读队列任务就会得到执行，从而读走数据，腾
            出空间 */
            vPrintString( "Could not send to the queue.\r\n" );
        }

        /* 让其他写队列任务得到执行。 */
        taskYIELD();
    }
}
```

程序清单 **38** 例 **11** 中写队列任务的实现代码.

读队列任务的优先级最低，所以只有在所有写队列任务都进入阻塞态后才有机会得到执行。而写队列任务只会在队列满时才会进入阻塞态，所以读队列任务得到执行时队列已满。因此读队列任务只管不停地读取数据，不必设定超时时间。

读队列任务的实现代码参见**程序列表 39**。

```
static void vReceiverTask( void *pvParameters )
{
/* 声明结构体变量以保存从队列中读出的数据单元 */
xData xReceivedStructure;
portBASE_TYPE xStatus;
    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* 读队列任务的优先级最低，所以其只可能在写队列任务阻塞时得到执行。而写队列任务只会在队列写
        满时才会进入阻塞态，所以读队列任务执行时队列肯定已满。所以队列中数据单元的个数应当等于队列的
        深度 – 本例中队列深度为3 */
        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\r\n" );
        }
        /* Receive from the queue.
        第二个参数是存放接收数据的缓存空间。本例简单地采用一个具有足够空间大小的变量的地址。

        第三个参数是阻塞超时时间 – 本例不需要指定超时时间，因为读队列任会只会在队列满时才会得到执行，
        故而不会因队列空而阻塞 */
        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );
        if( xStatus == pdPASS )
        {
            /* 数据成功读出，打印输出数值及数据来源。 */
            if( xReceivedStructure.ucSource == mainSENDER_1 )
            {
                vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
            }
            else
            {
                vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
        else
        {
            /* 没有读到任何数据。这一定是发生了错误，因为此任务只支在队列满时才会得到执行 */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
```

程序清单 **39** 例 **11** 中读队列任务的实现代码.

主函数 main()与上一例比起来只作了微小的改动。创建的队列数可以保存三个 xData 类型的数据单元，并且交换了写队列任务与读队列任务的优先级。本例 main()函数实现代码参见**程序清单 40**。

```
int main( void )
{
    /* 创建队列用于保存最多3个xData类型的数据单元。 */
    xQueue = xQueueCreate( 3, sizeof( xData ) );
    if( xQueue != NULL )
    {
        /* 为写队列任务创建2个实例。 The
        任务入口参数用于传递发送到队列中的数据。因此其中一个任务往队列中一直写入
        xStructsToSend[0]，而另一个则往队列中一直写入xStructsToSend[1]。这两个任务的优先级都
        设为2，高于读队列任务的优先级 */
        xTaskCreate( vSenderTask, "Sender1", 1000, &( xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &( xStructsToSend[ 1 ] ), 2, NULL );

        /* 创建读队列任务。
        读队列任务优先级设为1，低于写队列任务的优先级。 */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* 启动调度器，创建的任务得到执行。 */
        vTaskStartScheduler();
    }
    else
    {
        /* 创建队列失败。 */
    }

    /* 如果一切正常，main()函数不应该会执行到这里。但如果执行到这里，很可能是内存堆空间不足导致空闲
    任务无法创建。第五章将提供更多关于内存管理方面的信息 */
    for( ;; );
}
```

程序清单 40 例 11 的 main()函数实现代码。

和**例 10** 类似，写队列任务在每次循环中都主动进行任务切换，所以两个数据会被轮翻地写入到队列中。**图 24** 展示了**例 11** 代码运行的输出结果。

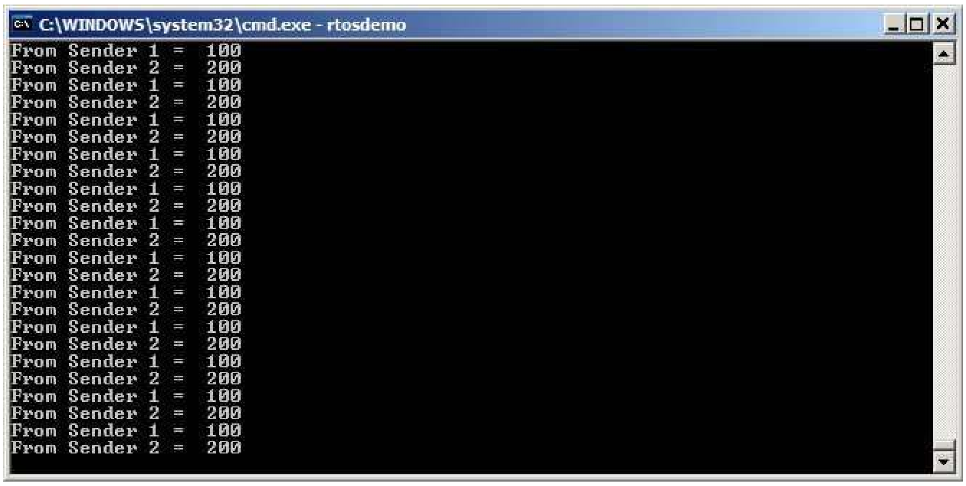


图 **24** 例 **11** 的输出结果

图 25 表述的是当写队列任务优先级高于读队列任务优先级时，各任务的执行顺序。对**图 25** 的更详细解释请参见**表 12**。

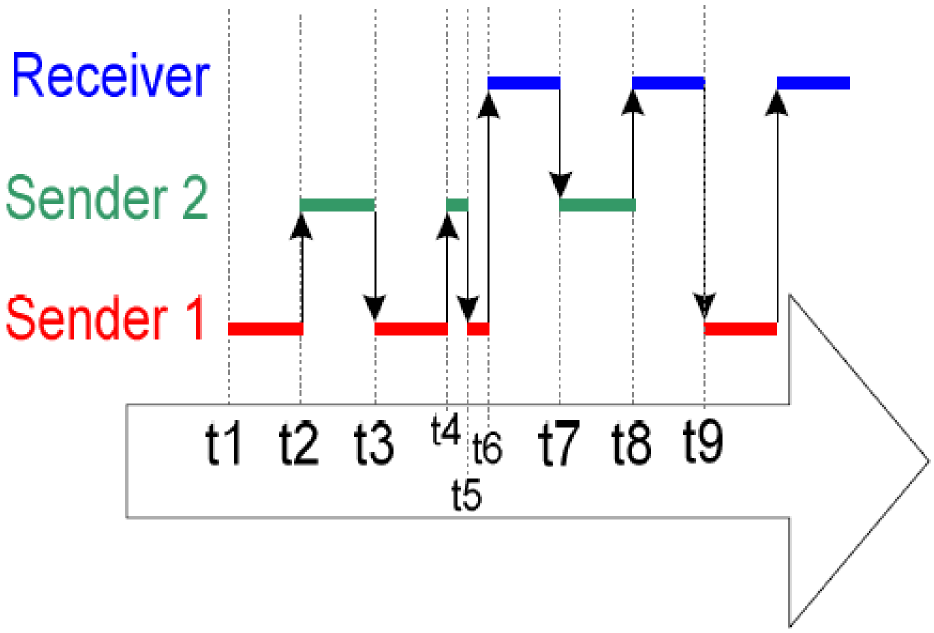


图 **25** 例 **11** 的执行流程

表 11 图 25 的要点解释

时刻	描述
t1	写队列任务 1 得到执行，并往队列中发送数据.
t2	写队列任务 1 切换到写队列任务 2。写队列任务 2 往队列中发送数据。
t3	写队列任务 2 又切回写队列任务 1。写队列任务 1 再次将数据写入队列，导致队列满。
t4	写队列任务 1 切换到写队列任务 2。
t5	写队列任务 2 试图往队列中写入数据。但由于队列已满，所以写队列任务 2 转入阻塞态以等待队列空间有效。这使得写队列任务 1 再次得到执行。
t6	写队列任务 1 试图往队列中写入数据。但由于队列已满，所以写队列任务 1 也转入阻塞态以等待队列空间有效。此时写队列任务均处于阻塞态，这才使得被赋予最低优先级的读队列任务得以执行。
t7	读队列任务从队列读取数据，并把读出的数据单元从队列中移出。一旦队列空间有效，写队列任务 2 立即解除阻塞，并且因为其具有更高优先级，所以抢占读队列任务。写队列任务 2 又往队列中写入数据，填充到刚刚被读队列任务腾出的存储空间，使得队列再一次变满。写队列发送完数据后便调用 taskYIELD()，但写队列任务 1 尚还处理阻塞态，所以写队列任务 2 并未被切换出去，继续执行。
t8	写队列任务 2 试图往队列中写入数据。但队列已满，所以写队列任务 2 转入阻塞态。两个写队列任务再一次同时处于阻塞态，所以读队列任务得以执行。
t9	读队列任务从队列读取数据，并把读出的数据单元从队列中移出。一旦队列空间有效，写队列任务 1 立即解除阻塞，并且因为其具有更高优先级，所以抢占读队列任务。写队列任务 1 又往队列中写入数据，填充到刚刚被读队列任务腾出的存储空间，使得队列再一次变满。写队列发送完数据后便调用 taskYIELD()，但写队列任务 2 尚还处理阻塞态，所以写队列任务 1 并未被切换出去，继续执行。写队列任务 1 试图往队列中写入数据。但队列已满，所以写队列任务 1 转入阻塞态。

两个写队列任务再一次同时处于阻塞态，所以读队列任务得以执行。

2.4 工作于大型数据单元

如果队列存储的数据单元尺寸较大，那最好是利用队列来传递数据的指针而不是对数据本身在队列上一字节一字节地拷贝进或拷贝出。传递指针无论是在处理速度上还是内存空间利用上都更有效。但是，当你利用队列传递指针时，一定要十分小心地做到以下两点：

1. 指针指向的内存空间的所有权必须明确

   当任务间通过指针共享内存时，应该从根本上保证所不会有任意两个任务同时修改共享内存中的数据，或是以其它行为方式使得共享内存数据无效或产生一致性问题。原则上，共享内存在其指针发送到队列之前，其内容只允许被发送任务访问；共享内存指针从队列中被读出之后，其内容亦只允许被接收任务访问。

2. 指针指向的内存空间必须有效

   如果指针指向的内存空间是动态分配的，只应该有一个任务负责对其进行内存释放。当这段内存空间被释放之后，就不应该有任何一个任务再访问这段空间。

切忌用指针访问任务栈上分配的空间。因为当栈帧发生改变后，栈上的数据将不再有效。

第三章

中断管理

3.1 概览

事件

嵌入式实时系统需要对整个系统环境产生的事件作出反应。举个例子，以太网外围部件收到了一个数据包(事件)，需要送到 TCP/IP 协议栈进行处理(反应)。更复杂的系统需要处理来自各种源头产生的事件，这些事件对处理时间和响应时间都有不同的要求。在各种情况下，都需要作出合理的判断，以达到最佳事件处理的实现策略：

1. 事件如何被检测到？通常采用中断方式，但是事件输入也可以通过查询获得。
2. 什么时候采用中断方式？中断服务例程(ISR)中的处理量有多大？以及 ISR 外的任务量有多大？通常情况下，ISR 应当越短越好。
3. 事件如何通知到主程序(这里指非 ISR 程序，而非 main()程序)代码？这些代码要如何架构才能最好地适应异步处理？

FreeRTOS 并没有为设计人员提供具体的事件处理策略，但是提供了一些特性使得设计人员采用的策略可以得到实现，而实现方式不仅简单，而且具有可维护性。

必须说明的是，只有以"FromISR"或"FROM_ISR"结束的 API 函数或宏才可以在中断服务例程中。

本章期望能清晰地告诉读者以下事情:

- 哪些 FreeRTOS 的 API 函数可以在中断服务例程中使用。
- 延迟中断方案是处何实现的。
- 如何创建和使用二值信号量以及计数信号量。
- 二值信号量和计数信号量之间的区别。
- 如何利用队利在中断服务例程中把数据传入传出。
- 一些 FreeRTOS 移植中采用的中断嵌套模型。

3.2 延迟中断处理

采用二值信号量同步

二值信号量可以在某个特殊的中断发生时，让任务解除阻塞，相当于让任务与中断同步。这样就可以让中断事件处理量大的工作在同步任务中完成，中断服务例程(ISR)中只是快速处理少部份工作。如此，中断处理可以说是被"推迟(deferred)"到一个"处理(handler)"任务。

如果某个中断处理要求特别紧急，其延迟处理任务的优先级可以设为最高，以保证延迟处理任务随时都抢占系统中的其它任务。这样，延迟处理任务就成为其对应的 ISR 退出后第一个执行的任务，在时间上紧接着 ISR 执行，相当于所有的处理都在 ISR 中完成一样。这种方案在图 **26** 中展现。

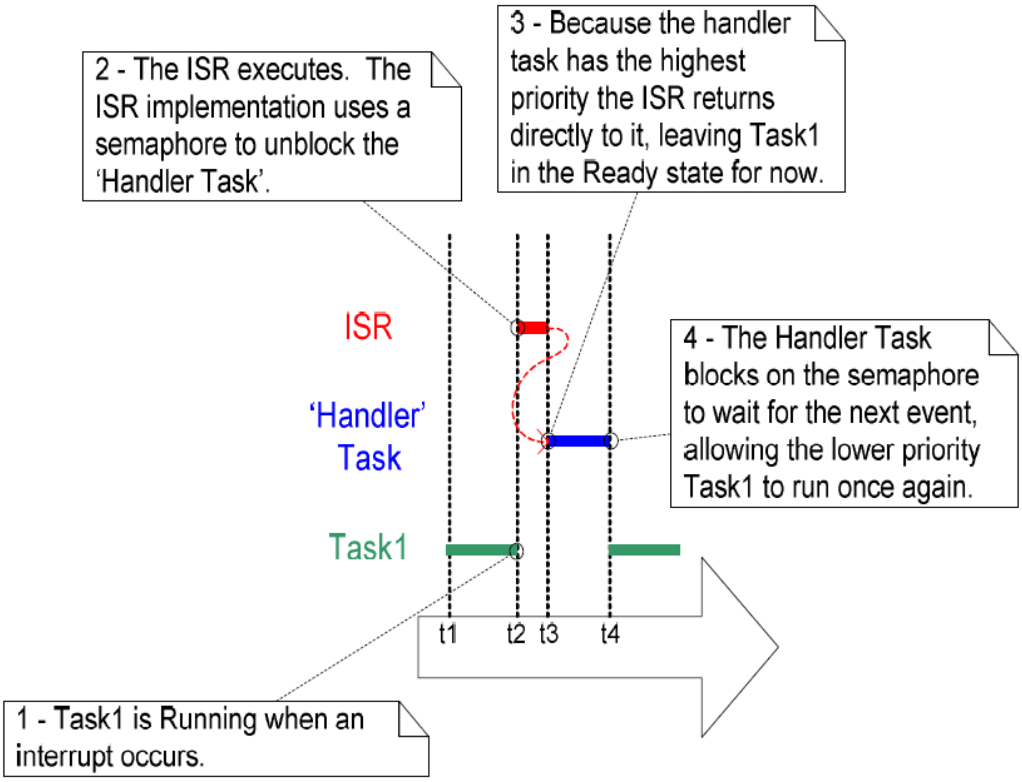


图 **26** 中断打断某个任务，但返回到另一个任务

延迟处理任务对一个信号量进行带阻塞性质的"take"调用，意思是进入阻塞态以等待事件发生。当事件发生后，ISR 对同一个信号量进行"give"操作，使得延迟处理任务解除阻塞，从而事件在延迟处理任务中得到相应的处理。

"获取(Taking，带走，按通常的说法译为获取)"和"给出(Giving)"信号量从概念上讲，

在不同的应用场合有不同的含义。在经典的信号量术语中，获取信号量等同于一个 P()操作，而给出信号量等同于一个 V()操作。

> 译者注： P 源自荷兰语 Parsseren，即英语的 Pass；V 源自荷兰语 Verhoog，即英语的 Increment。P(S)/V(S)操作是信号量的两个**原子操作**，S 为信号量 Semaphore，相当于一个标志，可以代表一个资源，一个事件等等，初始值视应用场合而定。P(S)/V(S) **原子操作**有如下行为：
>
> P(S)： IF (S <= 0) THEN 将本线程加入 S 的等待队列
>
> S = S – 1
>
> V(S)： S = S + 1
>
> IF (S > 0) THEN 唤醒某个等待线程

在这种中断同步的情形下，信号量可以看作是一个深度为 1 的队列。这个队列由于最多只能保存一个数据单元，所以其不为空则为满(所谓"二值")。延迟处理任务调用 xSemaphoreTake()时，等效于带阻塞时间地读取队列，如果队列为空的话任务则进入阻塞态。当事件发生后，ISR 简单地通过调用 xSemaphoreGiveFromISR()放置一个令牌(信号量)到队列中，使得队列成为满状态。这也使得延迟处理任务切出阻塞态，并移除令牌，使得队列再次成为空。当任务完成处理后，再次读取队列，发现队列为空，又进入阻塞态，等待下一次事件发生。整个流程在**图 27** 中有所展现。

如**图 27** 所示，中断给出信号量，甚至是在信号量第一次被获取之前就给出；而任务在获取信号量之后再也不给回来。这就是为什么说这种情况与读写队列相似。这也经常会给大家造成迷惑，因为这种情形和其它信号量的使用场合大不相同。在其它场合下，任务获得(Take)了信号量之后，必须得给(Give)回来——如同第四章描述一样。

vSemaphoreCreateBinary() API 函数

FreeRTOS 中各种信号量的句柄都存储在 xSemaphoreHandle 类型的变量中。

在使用信号量之前，必须先创建它。创建二值信号量使用 vSemaphoreCreateBinary()API 函数[2]

```
void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

程序清单 41 vSemaphoreCreateBinary() API 函数原型

表 41 vSemaphoreCreateBinary()参数

参数名	描述
xSemaphore	创建的信号量 需要说明的是 vSemaphoreCreateBinary()在实现上是一个宏，所以信号量变量应当直接传入，而不是传址。本章中包含本函数调用的示例可用于参考进行复制。

[2] 信号量 API 实际上是由一组宏实现的，而不是函数。本书中提及到这些宏的地方都简单地以函数相称

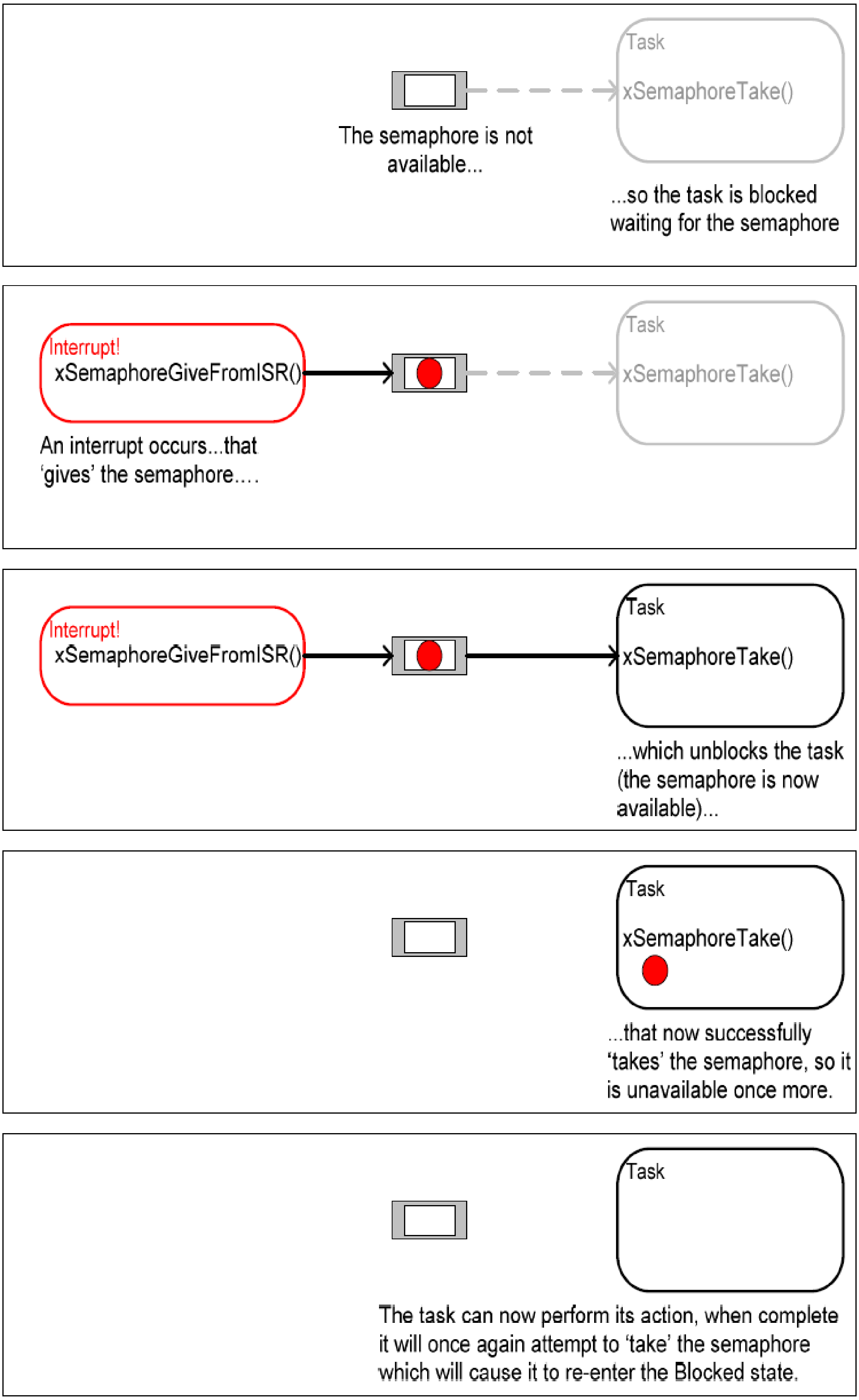


图 **27** 使用一个二值信号量实现任务与中断同步

xSemaphoreTake() API 函数

“带走(Taking)”一个信号量意为”获取(Obtain)”或”接收(Receive)”信号量。只有当信号量有效的时候才可以被获取。在经典信号量术中，xSemaphoreTake()等同于一次 P()操作。

除互斥信号量(Recursive Semaphore，直译为递归信号量，按通常的说法译为互斥信号量)外，所有类型的信号量都可以调用函数 xSemaphoreTake()来获取。

但 xSemaphoreTake()不能在中断服务例程中调用。

```
portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickType xTicksToWait );
```

程序清单 42 xSemaphoreTake() API 函数原型

表 13 xSemaphoreTake()参数及返回值

参数名	描述
xSemaphore	获取得到的信号量 信号量由定义为 xSemaphoreHandle 类型的变量引用。信号量在使用前必须先创建。
xTicksToWait	阻塞超时时间。任务进入阻塞态以等待信号量有效的最长时间。 如果 xTicksToWait 为 0，则 xSemaphoreTake()在信号量无效时会立即返回。 阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位。 如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制。

返回值　　有两个可能的返回值:

1. pdPASS

只有一种情况会返回 pdPASS，那就是成功获得信号量。

如果设定了阻塞超时时间(xTicksToWait 非 0)，在函数返回之前任务将被转移到阻塞态以等待信号量有效。如果在超时到来前信号量变为有效，亦可被成功获取，返回 pdPASS。

2. pdFALSE

未能获得信号量。

如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待信号量有效。但直到超时信号量也没有变为有效，所以不会获得信号量，返回 pdFALSE。

xSemaphoreGiveFromISR() API 函数

除互斥信号量外，FreeRTOS 支持的其它类型的信号量都可以通过调用 xSemaphoreGiveFromISR()给出。

xSemaphoreGiveFromISR()是 xSemaphoreGive()的特殊形式，专门用于中断服务例程中。

```
portBASE_TYPE xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore,
                                     portBASE_TYPE *pxHigherPriorityTaskWoken );
```

程序清单 43　xSemaphoreGiveFromISR() API 函数原型

表 14 xSemaphoreGiveFromISR()参数与返回值

参数名	描述
xSemaphore	给出的信号量 信号量由定义为 xSemaphoreHandle 类型的变量引用。信号量在使用前必须先创建。
pxHigherPriorityTaskWoken	对某个信号量而言，可能有不止一个任务处于阻塞态在等待其有效。调用 xSemaphoreGiveFromISR()会让信号量变为有效，所以会让其中一个等待任务切出阻塞态。如果调用 xSemaphoreGiveFromISR()使得一个任务解除阻塞，并且这个任务的优先级高于当前任务(也就是被中断的任务)，那么 xSemaphoreGiveFromISR()会在函数内部将 *pxHigherPriorityTaskWoken 设为 pdTRUE。 如果 xSemaphoreGiveFromISR() 将此值设为 pdTRUE，则在中断退出前应当进行一次上下文切换。这样才能保证中断直接返回到就绪态任务中优先级最高的任务中。
返回值	有两个可能的返回值: 1. pdPASS xSemaphoreGiveFromISR()调用成功。 2. pdFAIL 如果信号量已经有效，无法给出，则返回 pdFAIL。

例 12. 利用二值信号量对任务和中断进行同步

本例在中断服务例程中使用一个二值信号量让任务从阻塞态中切换出来——从效果上等同于让任务与中断进行同步。

一个简单的周期性任务用于每隔 500 毫秒产生一个软件中断。之所以采用软件中断，是因为在模拟的 DOS 环境中，很难挂接一个真正的 IRQ 中断。相比之下，使用软件中断要方便得多。**程序清单 44** 即是这个周期任务的实现代码。需要说明的是，此任务在产生中断之前和之后都会打印输出一个字符串。这样就可以在最终的执行结果中直观地看出整个程序的执行流程。

```
static void vPeriodicTask( void *pvParameters )
{
    for( ;; )
    {
        /* 此任务通过每500毫秒产生一个软件中断来"模拟"中断事件 */
        vTaskDelay( 500 / portTICK_RATE_MS );
        /* 产生中断，并在产生之前和之后输出信息，以便在执行结果中直观直出执行流程 */
        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );
        __asm{ int 0x82 } /* 这条语句产生中断 */
        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

程序清单 44 例12中用于周期性产生软件中断的周期任务实现代码

程序清单 45 展现的是延迟处理任务的具体实现——此任务通过使用二值信号量与软件中断进行同步。这个任务也在每次循环中打印输出一个信息，这样做的目的同样是可以在程序的执行输出结果中直观地看出任务与中断的执行流程。

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* 使用信号量等待一个事件。信号量在调度器启动之前，也即此任务执行之前就已被创建。任务被无超
        时阻塞，所以此函数调用也只会在成功获取信号量之后才会返回。此处也没有必要检测返回值 */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

        /* 程序运行到这里时，事件必然已经发生。本例的事件处理只是简单地打印输出一个信息 */
        vPrintString( "Handler task - Processing event.\r\n" );
    }
}
```

程序清单 45 例 12 中延迟处理任务的实现代码(此任务与中断同步)

程序清单 46 展现的是中断服务例程，这才是真正的中断处理程序。这段代码做的事情非常少，仅仅是给出一个信号量，以让延迟处理任务解除阻塞。注意这里是如何使用参数 pxHigherPriorityTaskWoken 的。这个参数在调用 xSemaphoreGiveFromISR() 前被设置为 pdFALSE，如果在调用完成后被置为 pdTRUE，则需要进行一次上下文切换。

本例中断服务例程的语法，以及用于上下文切换调用的宏，都是基于 Open Watcom DOS 平台的移植，与其它平台的移植可能会有所不同。对于实际使用的平台，请参考对应移植的 demo 应用示例，以找到正确的语法要求。

```
static void __interrupt __far vExampleInterruptHandler( void )
{
static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;

    /* 'Give' the semaphore to unblock the task. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* 给出信号量以使得等待此信号量的任务解除阻塞。如果解出阻塞的任务的优先级高于当前任务的优先
        级 – 强制进行一次任务切换，以确保中断直接返回到解出阻塞的任务(优选级更高)。

        说明：在实际使用中，ISR中强制上下文切换的宏依赖于具体移植。此处调用的是基于Open Watcom DOS
        移植的宏。其它平台下的移植可能有不同的语法要求。对于实际使用的平台，请参如数对应移植自带的示
        例程序，以决定正确的语法和符号。
        */
        portSWITCH_CONTEXT();
    }
}
```

程序清单 46　例 12 中软件中断的中断服务例程

main()函数很简单，创建二值信号量及任务，安装中断服务例程，然后启动调度器。具体实现参见**程序清单 47**。

```
int main( void )
{
    /* 信号量在使用前都必须先创建。本例中创建了一个二值信号量 */
    vSemaphoreCreateBinary( xBinarySemaphore );

    /* 安装中断服务例程 */
    _dos_setvect( 0x82, vExampleInterruptHandler );

    /* 检查信号量是否成功创建 */
    if( xBinarySemaphore != NULL )
    {
        /* 创建延迟处理任务。此任务将与中断同步。延迟处理任务在创建时使用了一个较高的优先级，以保证
        中断退出后会被立即执行。在本例中，为延迟处理任务赋予优先级3 */
        xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );

        /* 创建一个任务用于周期性产生软件中断。此任务的优先级低于延迟处理任务。每当延迟处理任务切出
        阻塞态，就会抢占周期任务*/
        xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }

    /* 如果一切正常，main()函数不会执行到这里，因为调度器已经开始运行任务。但如果程序运行到了这里，
    很可能是由于系统内存不足而无法创建空闲任务。第五章会提供更多关于内存管理的信息 */
    for( ;; );
}
```

程序清单 47　例 12 中的 main()函数实现代码

例 **12** 的输出结果参见图 **28**。和期望的一样，延迟处理任务在中断产生后立即执行。所以延迟处理任务的输出信息将周期任务的两条输出信息分开。图 **29** 对执行流程作出了进一步的解释。

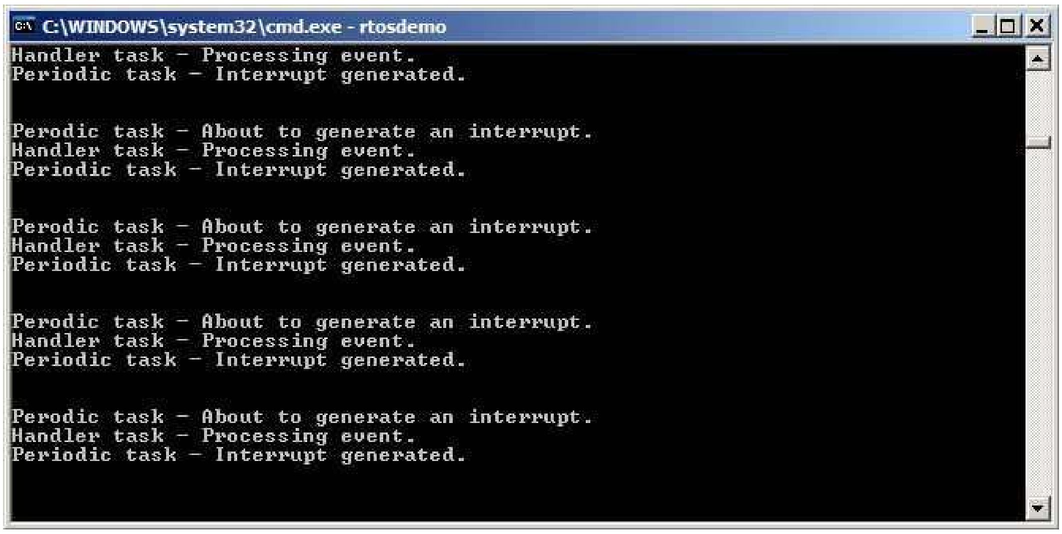


图 28 例 12 代码执行的输出结果

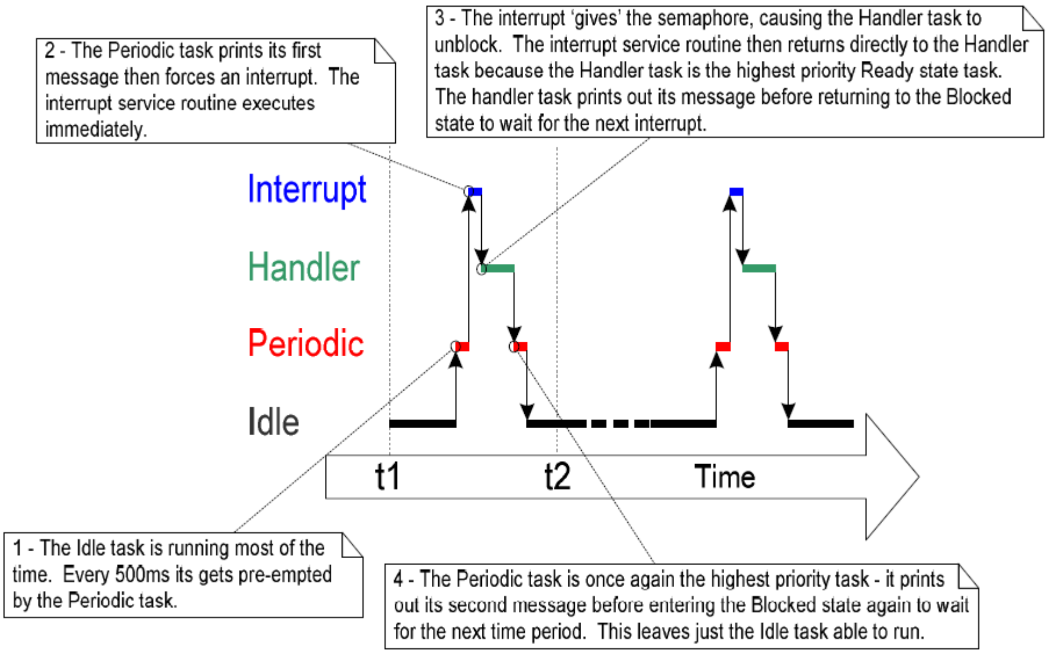


图 29 例 12 中代码的执行流程

3.3 计数信号量

例 12 演示了一个二值信号量被用于让任务和中断进行同步。整个执行流程可以描述为:

1. 中断产生。
2. 中断服务例程启动，给出信号量以使延迟处理任务解除阻塞。
3. 当中断服务例程退出时，延迟处理任务得到执行。延迟处理任务做的第一件事便是获取信号量。
4. 延迟处理任务完成中断事件处理后，试图再次获取信号量——如果此时信号量无效，任务将切入阻塞待等待事件发生。

在中断以相对较慢的频率发生的情况下，上面描述的流程是足够而完美的。如果在延迟处理任务完成上一个中断事件的处理之前，新的中断事件又发生了，等效于将新的事件锁存在二值信号量中，使得延迟处理任务在处理完上一个事件之后，立即就可以处理新的事件。也就是说，延迟处理任务在两次事件处理之间，不会有进入阻塞态的机会，因为信号量中锁存有一个事件，所以当 xSempaphoreTake()调用时，信号量立即有效。这种情形将在**图 30** 中进行展现。

在**图 30** 中可以看到，一个二值信号量最多只可以锁存一个中断事件。在锁存的事件还未被处理之前，如果还有中断事件发生，那么后续发生的中断事件将会丢失。如果用计数信号量代替二值信号量，那么，这种丢中断的情形将可以避免。

就如同我们可以把二值信号量看作是只有一个数据单元的队列一样，计数信号量可以看作是深度大于 1 的队列。任务其实对队列中存储的具体数据并不感兴趣——其只关心队列是空还是非空。

计数信号量每次被给出(Given)，其队列中的另一个空间将会被使用。队列中的有效数据单元个数就是信号量的"计数(Count)"值。

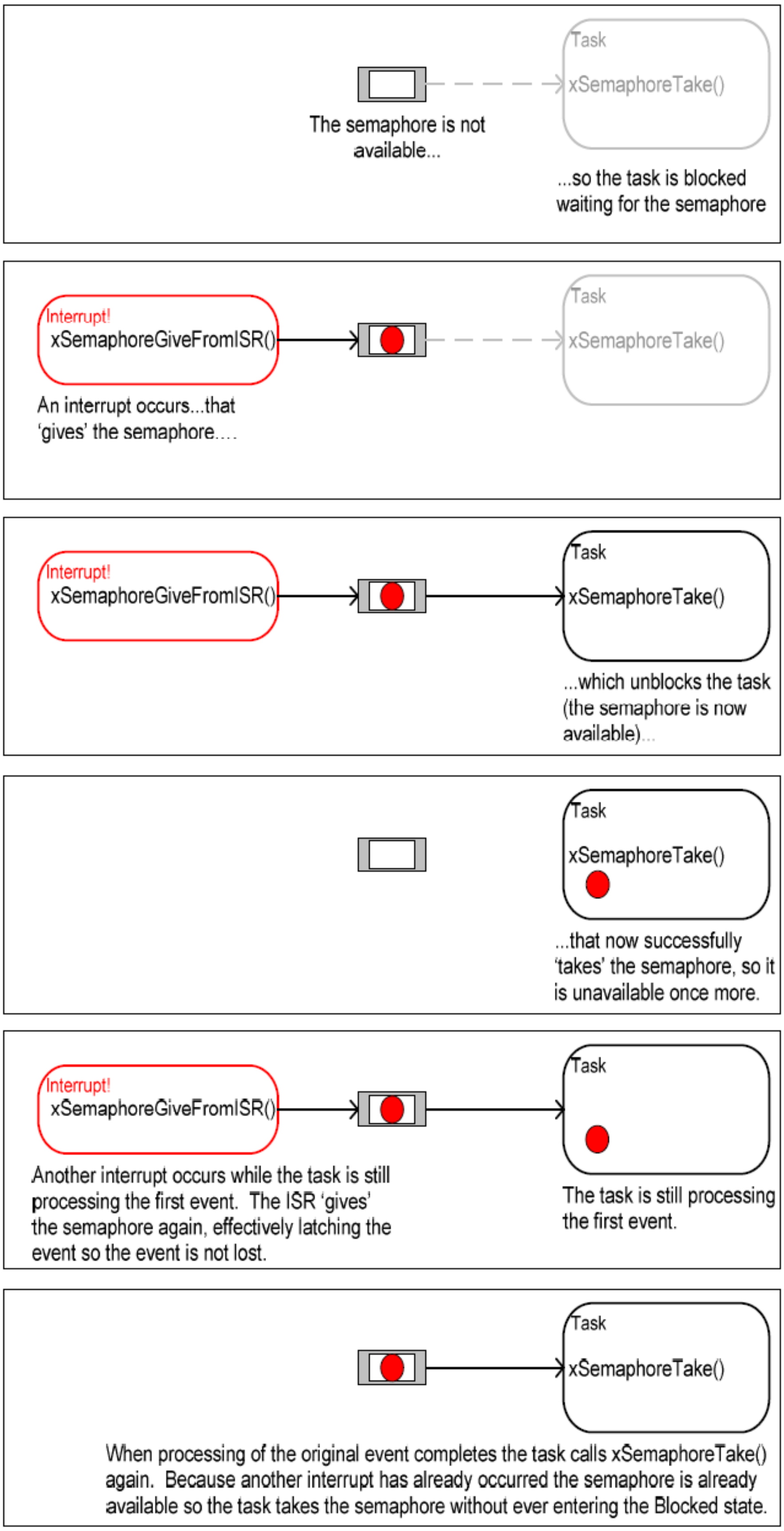


图 30 一个二值信号量最多只能锁存一个中断事件

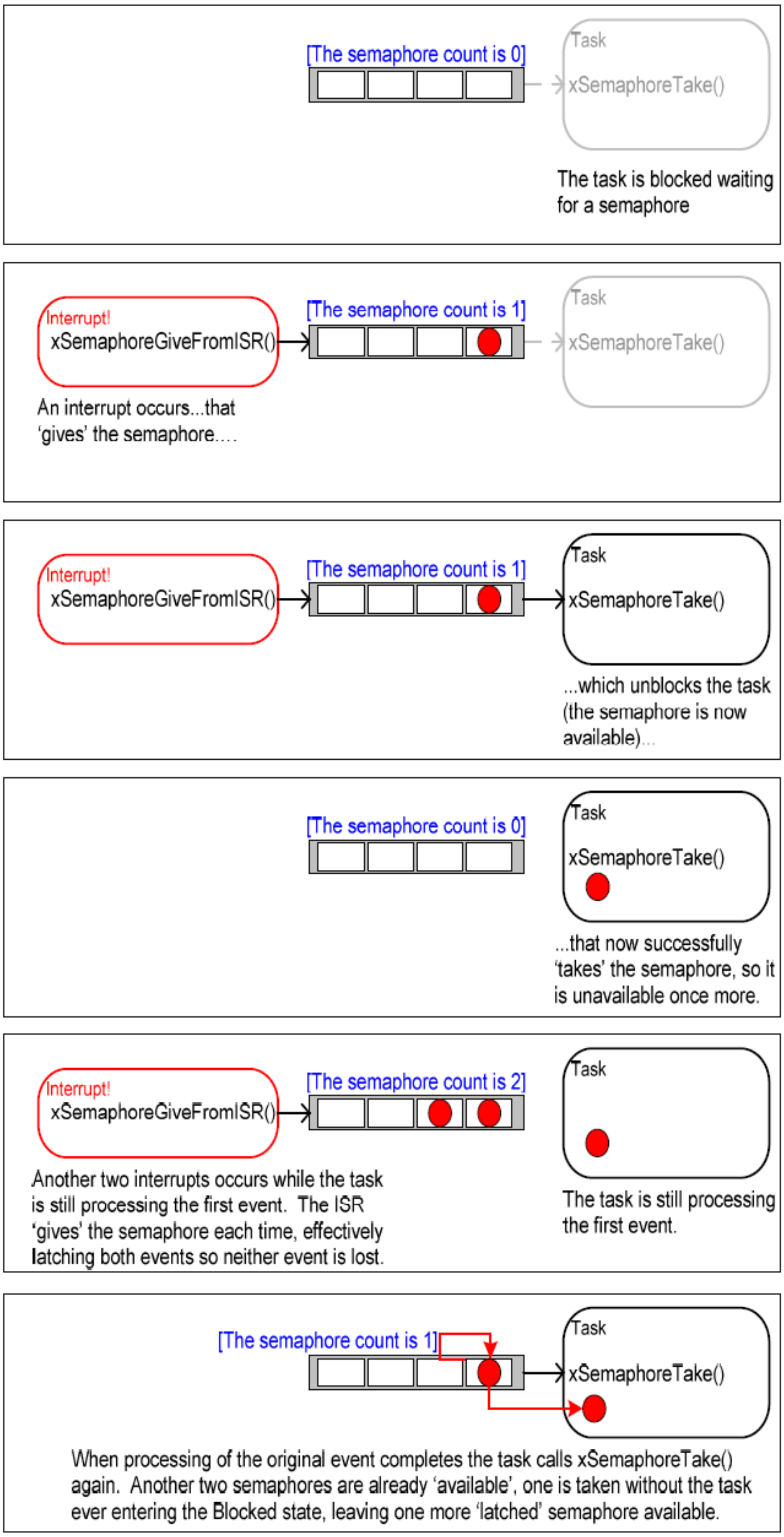


图 **31** 使用计数信号量对事件 **"计数(Count)"**

计数信号量有以下两种典型用法：

1. 事件计数

在这种用法中，每次事件发生时，中断服务例程都会“给出(Give)”信号量——信号量在每次被给出时其计数值加 1。延迟处理任务每处理一个任务都会”获取(Take)”一次信号量——信号量在每次被获取时其计数值减 1。信号量的计数值其实就是已发生事件的数目与已处理事件的数目之间的差值。这种机制可以参考图 31。

用于事件计数的计数信号量，在被创建时其计数值被初始化为 0。

2. 资源管理

在这种用法中，信号量的计数值用于表示可用资源的数目。一个任务要获取资源的控制权，其必须先获得信号量——使信号量的计数值减 1。当计数值减至 0，则表示没有可用资源。当任务利用资源完成工作后，将给出(归还)信号量——使信号量的计数值加 1。

用于资源管理的信号量，在创建时其计数值被初始化为可用资源总数。第四章涵盖了使用信号量来管理资源。

xSemaphoreCreateCounting() API 函数

FreeRTOS 中所有种类的信号量句柄都由声明为 xSemaphoreHandle 类型的变量保存。

信号量在使用前必须先被创建。使用 xSemaphoreCreateCounting() API 函数来创建一个计数信号量。

```
xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_TYPE uxMaxCount,
                                           unsigned portBASE_TYPE uxInitialCount );
```

程序清单 48 xSemaphoreCreateCounting() API 函数原型

表 15　xSemaphoreCreateCounting()参数与返回值

参数名	描述
uxMaxCount	最大计数值。如果把计数信号量类比于队列的话，uxMaxCount 值就是队列的最大深度。 当此信号量用于对事件计数或锁存事件的话，uxMaxCount 就是可锁存事件的最大数目。 当此信号量用于对一组资源的访问进行管理的话，uxMaxCount 应当设置为所有可用资源的总数。
uxInitialCount	信号量的初始计数值。 当此信号量用于事件计数的话，uxInitialCount 应当设置为 0——因为当信号量被创建时，还没有事件发生。 当此信号量用于资源管理的话，uxInitialCount 应当等于 uxMaxCount——因为当信号量被创建时，所有的资源都是可用的。
返回值	如果返回 NULL 值，表示堆上内存空间不足，所以 FreeRTOS 无法为信号量结构分配内存导致信号量创建失败。第五章有提供更多的内存管理方面的信息。 如果返回非 NULL 值，则表示信号量创建成功。此值应当被保存起来作为这个的信号量的句柄。

例 13. 利用计数信号量对任务和中断进行同步

例 13 用计数信号量代替二值信号量对**例 12** 的实现进行了改进。修改 main()函数调用 xSemaphoreCreateCounting()，以代替对 xSemaphoreCreateBinary()的调用。新的 API 调用如程序清单 49 所示：

```
/* 在信号量使用之前必须先创建。本例中创建了一个计数信号量。此信号量的最大计数值为10，初始计数值为0 */
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

程序清单 49 使用 xSemaphoreCreateCounting() 创建一个计数信号量

为了模拟多个事件以高频率发生，修改了中断服务例程，在每次中断多次"给出(Give)"信号量。每个事件都锁存到信号量的计数值中。修改后的中断服务例程如**程序清单 50** 所示。

```
static void __interrupt __far vExampleInterruptHandler( void )
{
static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;

    /* 多次给出信号量。第一次给出时使得延迟处理任务解除阻塞。后续给出用于演示利用被信号量锁存事件，
    以便延迟处理任何依序对这些中断事件进行处理而不会丢中断。用这种方式来模拟处理器产生多个中断，尽管
    这些事件只是在单次中断中模拟出来的 */
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* 给出信号量以使得等待此信号量的任务解除阻塞。如果解出阻塞的任务的优先级高于当前任务的优先
        级 – 强制进行一次任务切换，以确保中断直接返回到解出阻塞的任务(优选级更高)。

        说明：在实际使用中，ISR中强制上下文切换的宏依赖于具体移植。此处调用的是基于Open Watcom DOS
        移植的宏。其它平台下的移植可能有不同的语法要求。对于实际使用的平台，请参如数对应移植自带的示
        例程序，以决定正确的语法和符号。
        */
        portSWITCH_CONTEXT();
    }
}
```

程序清单 50 例 13 中的中断服务例程实现代码

其它函数都复用**例 12** 中的代码，保持不变。

图 32 展示了**例 13** 的输出结果。从图中可以看到，每次中断发生后，延迟处理任务处理了中断生成的全部三个事件[模拟出来的]。这些事件被锁存到信号量的计数值中，以使得延迟处理任务可以对它们依序进行处理。



图 32 例 13 的输出结果

3.4 在中断服务例程中使用队列

xQueueSendToFrontFromISR()，xQueueSendToBackFromISR()与 xQueueReceiveFromISR()分别是 xQueueSendToFront()，xQueueSendToBack()与 xQueueReceive()的中断安全版本，专门用于中断服务例程中。

信号量用于事件通信。而队列不仅可以用于事件通信，还可以用来传递数据。

xQueueSendToFrontFromISR()与 xQueueSendToBackFromISR() API 函数

xQueueSendFromISR()完全等同于 xQueueSendToBackFromISR()。

```
portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle xQueue,
                                        void *pvItemToQueue
                                        portBASE_TYPE *pxHigherPriorityTaskWoken );
```

程序清单 51 xQueueSendToFrontFromISR() API 函数原型

```
portBASE_TYPE xQueueSendToBackFromISR( xQueueHandle xQueue,
                                       void *pvItemToQueue
                                       portBASE_TYPE *pxHigherPriorityTaskWoken
);
```

程序清单 52 xQueueSendToBackFromISR() API 函数原型

表 16 xQueueSendToFrontFromISR 与 xQueueSendToBackFromISR()参数与返回值

参数名	描述
xQueue	目标队列的句柄。这个句柄即是调用 xQueueCreate() 创建该队列时的返回值。
pvItemToQueue	发送数据的指针。其指向将要复制到目标队列中的数据单元。 由于在创建队列时设置了队列中数据单元的长度，所以会从该指针指向的空间复制对应长度的数据到队列的存储区域。

pxHigherPriorityTaskWoken	对某个队列而言，可能有不止一个任务处于阻塞态在等待其数据有效。调用 xQueueSendToFrontFromISR()或 xQueueSendToBackFromISR()会使得队列数据变为有效，所以会让其中一个等待任务切出阻塞态。如果调用这两个 API 函数使得一个任务解除阻塞，并且这个任务的优先级高于当前任务(也就是被中断的任务)，那么 API 会在函数内部将*pxHigherPriorityTaskWoken 设为 pdTRUE。 如果这两个 API 函数将此值设为 pdTRUE，则在中断退出前应当进行一次上下文切换。这样才能保证中断直接返回到就绪态任务中优先级最高的任务中。
返回值	有两个可能的返回值: 1. pdPASS 返回 pdPASS 只会有一种情况，那就是数据被成功发送到队列中。 2. errQUEUE_FULL 如果由于队列已满而无法将数据写入，则将返回 errQUEUE_FULL。

有效使用队列

FreeRTOS 的大多数 demo 应用程序中都包含一个简单的 UART 驱动，其通过队列将字符传递到发送中断例程，也使用队列将字符从接收中断例程中传递出来。发送或接收的每个字符都通过队列单独传递。这些 UART 驱动的这种实现方式只是单纯了为了演示如何在中断中使用队列。实际上利用队列传递单个字符是极其低效的，特别是在波特率较高的时后，所以这种方式并不建议用在产品代码中。实际应用中可以采用下述

更有效的方式：

- 将接收到的字符先缓存到内存中。当接收到一个传输完成消息，或是检测到传输中断后，使用信号量让某个任务解除阻塞，这个任务将对字符缓存进行处理。
- 在中断服务中直接解析接收到的字符，然后通过队列将解析后经解码得到的命令发送到处理任务(与图 23 中描述的方式类似)。这种技术仅适用于数据流能够快速解析的场合，这样整个数据解析工作才可以放在中断服务中完成。

例 14. 利用队列在中断服务中发送或接收数据

本例演示在同一个中断服务中使用 xQueueSendToBackFromISR() 和 xQueueReceiveFromISR()。和之前一样，采用软件中断以方便实现。

创建一个周期任务用于每 200 毫秒往队列中发送五个数值，五个数值都发送完后便产生一个软件中断。周期任务的实现代码参见**程序清单 53**。

```
static void vIntegerGenerator( void *pvParameters )
{
portTickType xLastExecutionTime;
unsigned portLONG ulValueToSend = 0;
int i;
    /* 初始化变量，用于调用 vTaskDelayUntil(). */
    xLastExecutionTime = xTaskGetTickCount();
    for( ;; )
    {
        /* 这是个周期性任务。进入阻塞态，直到该再次运行的时刻。此任务每200毫秒执行一次 */
        vTaskDelayUntil( &xLastExecutionTime, 200 / portTICK_RATE_MS );
        /* 连续五次发送递增数值到队列。这此数值将在中断服务例程中读出。中断服务例程会将队列读空，所
        以此任务可以确保将所有的数值都发送到队列。因此不需要指定阻塞超时时间 */
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++;
        }
        /* 产生中断，以让中断服务例程读取队列 */
        vPrintString( "Generator task - About to generate an interrupt.\r\n" );
        __asm{ int 0x82 } /* This line generates the interrupt. */
        vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

程序清单 53 例 14 中的写队列任务实现代码

中断服务例程重复调用 xQueueReceiveFromISR()，直到被周期任务写到队列的数值都被读出，以将队列读空。每个接收到的数值的低两位用于一个字符串数组的索引，被索引到的字符串的指针将通过调用 xQueueSendFromISR()发送到另一个队列中。中断服务例程的实现代码参数**程序清单 54**。

```
static void __interrupt __far vExampleInterruptHandler( void )
{
static portBASE_TYPE xHigherPriorityTaskWoken;
static unsigned long ulReceivedNumber;

/* 这些字符串被声明为static const，以保证它们不会被定位到ISR的栈空间中，即使ISR没有运行它们也是存在的 */
static const char *pcStrings[] =
{
    "String 0\r\n",
    "String 1\r\n",
    "String 2\r\n",
    "String 3\r\n"
};
    xHigherPriorityTaskWoken = pdFALSE;

    /* 重复执行，直到队列为空 */
    while( xQueueReceiveFromISR( xIntegerQueue,
                                 &ulReceivedNumber,
                                 &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
    {
        /* 截断收到的数据，保留低两位(数值范围0到3).然后将索引到的字符串指针发送到另一个队列 */
        ulReceivedNumber &= 0x03;
        xQueueSendToBackFromISR( xStringQueue,
                                 &pcStrings[ ulReceivedNumber ],
                                &xHigherPriorityTaskWoken );
    }

    /* 被队列读写操作解除阻塞的任务，其优先级是否高于当前任务？如果是，则进行任务上下文切换 */
    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* 说明：在实际使用中，ISR中强制上下文切换的宏依赖于具体移植。此处调用的是基于Open Watcom DOS移植的宏。其它平台下的移植可能有不同的语法要求。对于实际使用的平台，请参如数对应移植自带的示例程序，以决定正确的语法和符号。 */
        portSWITCH_CONTEXT();
    }
}
```

程序清单 **54** 例 **14** 的中断服务例程实现代码

另一个任务将接收从中断服务例程发出的字符串指针。此任务在读队列时被阻塞，直到队列中有消息到来，并将接收到的字符串打印输出。其实现代码参见**程序清单 55**。

```
static void vStringPrinter( void *pvParameters )
{
char *pcString;
    for( ;; )
    {
        /* Block on the queue to wait for data to arrive. */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

        /* Print out the string received. */
        vPrintString( pcString );
    }
}
```

程序清单 55　例 14 中的字符串接收任务实现，其接收来自中断服务例程的字符串，并打印输出

和往常一样，main()函数创建队列和任务，然后启动调度器。见**程序清单 56**。

```
int main( void )
{
    /* 队列使用前必须先创建。本例中创建了两个队列。一个队列用于保存类型为unsigned long的变量，另一
    个队列用于保存类型为char*的变量。两个队列的深度都为10。在实际应用中应当检测返回值以确保队列创建
    成功 */
    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* 安装中断服务例程。 */
    _dos_setvect( 0x82, vExampleInterruptHandler );

    /* 创建任务用于往中断服务例程中发送数值。此任务优先级为1 */
    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );
    /* 创建任务用于从中断服务例程中接收字符串，并打印输出。此任务优先级为2 */
    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );

    /* Start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();

    /* 如果一切正常，main()函数不会执行到这里，因为调度器已经开始运行任务。但如果程序运行到了这里，
    很可能是由于系统内存不足而无法创建空闲任务。第五章会提供更多关于内存管理的信息 */
    for( ;; );
}
```

程序清单 56　例 14 中的 main()函数实现

例 14 的运行输出参见图 33。从图中可以看到，中断服务例程接收到所有五个数值，并以五个字符串作为响应。更多解释请参考图 34

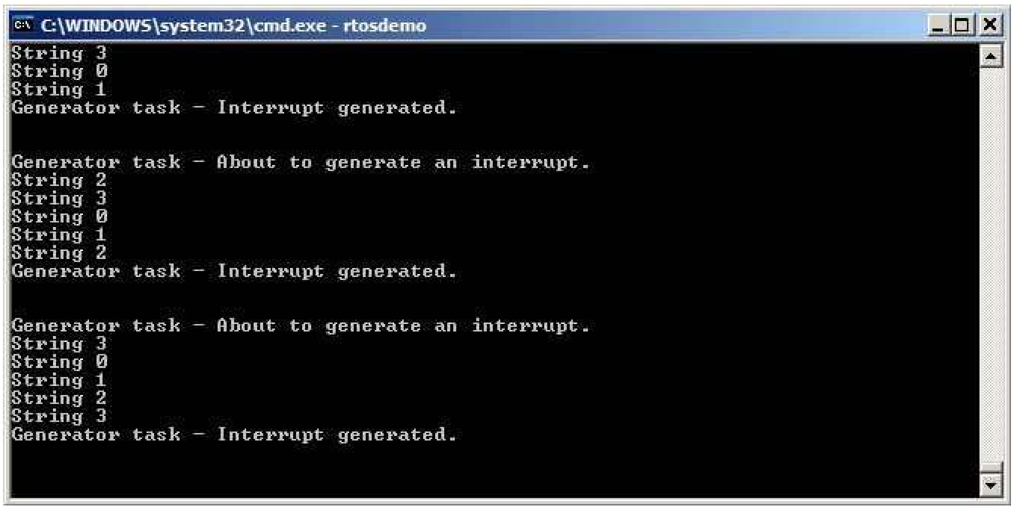


图 33 例 14 的运行输出

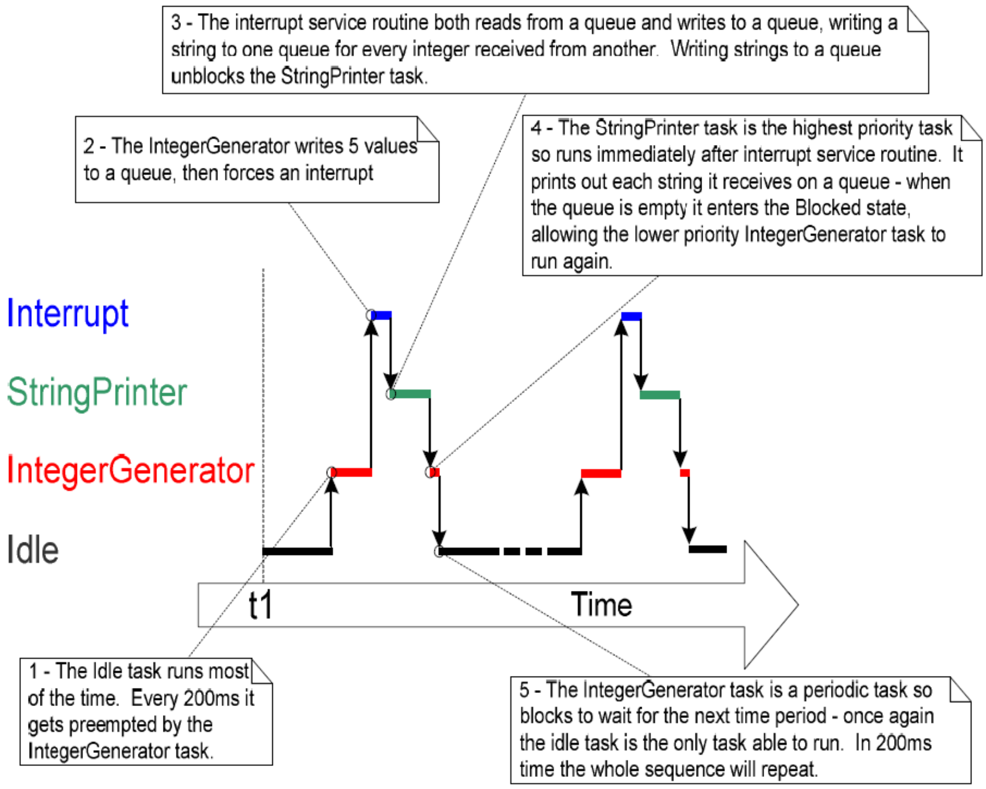


图 34 例 14 的执行流程

3.5 中断嵌套

最新的 FreeRTOS 移植中允许中断嵌套。中断嵌套需要在 FreeRTOSConfig.h 中定义**表 17** 详细列出的一个或两个常量。

表 17 控制中断嵌套的常量

常量	描述
configKERNEL_INTERRUPT_PRIORITY	设置系统心跳时钟的中断优先级。 如果在移植中没有使用常量 configMAX_SYSCALL_INTERRUPT_PRIORITY，那么需要调用中断安全版本 FreeRTOS API 的中断都必须运行在此优先级上。
configMAX_SYSCALL_INTERRUPT_PRIORITY	设置中断安全版本 FreeRTOS API 可以运行的最高中断优先级。

建立一个全面的中断嵌套模型需要设置 configMAX_SYSCALL_INTERRUPT_PRIRITY 为比 configKERNEL_INTERRUPT_PRIORITY 更高的优先级。这种模型在图 35 中有所展示。图 35 所示的情形假定常量 configMAX_SYSCALL_INTERRUPT_PRIRITY 设置为 3，configKERNEL_INTERRUPT_PRIORITY 设置为 1。同时也假定这种情形基于一个具有七个不同中断优先及的微控制器。这里的七个优先级仅仅是本例的一种假定，并非对应于任何一种特定的微控制器架构。

在任务优先级和中断优先级之间常常会产生一些混淆。图 35 所示的中断优先级是由微控制器架构体系所定义的。中断优先级是硬件控制的优先级，中断服务例程的执行会与之关联。任务并非运行在中断服务中，所以赋予任务的软件优先级与赋予中断源的硬件优先级之间没有任何关系。

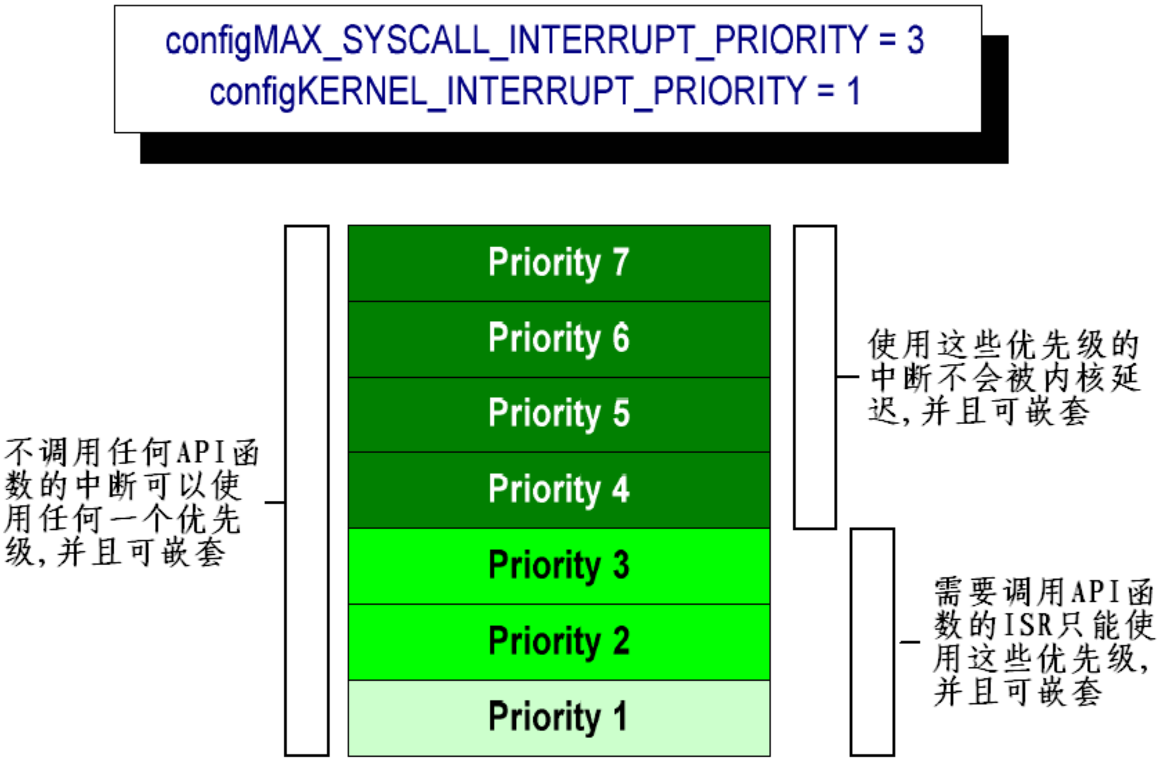


图 35　中断控制常量影响中断嵌套行为

如图 35 所示：

- 处于中断优先级 1 到 3(含)的中断会被内核或处于临界区的应用程序阻塞执行，但是它们可以调用中断安全版本的 FreeRTOS API 函数
- 处于中断优先级 4 及以上的中断不受临界区影响，所以其不会被内核的任何行为阻塞，可以立即得到执行——这是由微控制器本身对中断优先级的限定所决定的。通常需要严格时间精度的功能（如电机控制）会使用高于 configMAX_SYSCALL_INTERRUPT_PRIRITY 的优先级，以保证调度器不会对其中断响应时间造成抖动。
- 不需要调用任何 FreeRTOS API 函数的中断，可以自由地使用任意优先级。

对 ARM Cortex M3 用户的一点提示

Cortex M3 使用低优先级号数值表示逻辑上的高优先级中断。这显得不是那么直观，所以很容易被忘记。如果你想对某个中断赋予低优先级，则必须使用一个高优先级号数值。千万不要给它指定优先级号 0(或是其它低优先级号数值)，因为这将会使得这个中断在系统中拥有最高优先级——如果这个优先级高于 configMAX_SYSCALL_INTERRUPT_PRIRITY，将很可能导致系统崩溃。

Cortex M3 内核的最低优先级为 255，但是不同的 Cortex M3 处理器厂商实现的优先级位数不尽相同，而各自的配套库函数也使用了不同的方式来支持中断优先级。比如 STM32，ST 的驱动库中将最低优先级指定为 15，而最高优先级指定为 0。

第四章

资源管理

4.1 概览

多任务系统中存在一种潜在的风险。当一个任务在使用某个资源的过程中，即还没有完全结束对资源的访问时，便被切出运行态，使得资源处于非一致，不完整的状态。如果这个时候有另一个任务或者中断来访问这个资源，则会导致数据损坏或是其它相似的错误。

以下便是一些例子：

1. 访问外设

考虑如下情形，有两个任务都试图往一个 LCD 中写数据：

- 任务 A 运行，并往 LCD 写字符串"Hello world"。
- 任务 A 被任务 B 抢占，但此时字符串才输出到"Hello w"。
- 任务 B 往 LCD 写"Abort, Retry, Fail?"，然后进入阻塞态。
- 任务 A 从被抢占处继续执行，完成剩余的字符输出——"orld"。

现在 LCD 显示的是被破坏了的字符串"Hello wAbort, Retry, Fail?orld"。

2. 读-改-写操作

程序清单 57 展现的是一段 C 代码和其等效的 ARM7 汇编代码。可以看出，PORTA 中的值先从内存读到寄存器，在寄存器中完成修改，然后再写回内存。这所是所谓的读-改-写操作。

```
/* The C code being compiled. */
155: PORTA |= 0x01;

/* The assembly code produced. */
0x00000264 481C LDR R0,[PC,#0x0070]  ; Obtain the address of PORTA
0x00000266 6801 LDR R1,[R0,#0x00]    ; Read the value of PORTA into R1
0x00000268 2201 MOV R2,#0x01         ; Move the absolute constant 1 into R2
0x0000026A 4311 ORR R1,R2            ; OR R1 (PORTA) with R2 (constant 1)
0x0000026C 6001 STR R1,[R0,#0x00]    ; Store the new value back to PORTA
```

程序清单 57 读-改-写过程示例

这是一个"非原子"操作，因为完成整个操作需要不止一条指令，所以操作过程可能被中断。考虑如下情形，两个任务都试图更新一个名为 PORTA 的内存映射寄存器：

- 任务 A 把 PORTA 的值加载到寄存器中——整个流程的读操作。
- 在任务 A 完成整个流程的改和写操作之前，被任务 B 抢占。
- 任务 B 完整的执行了对 PORTA 的更新流程，然后进入阻塞态。
- 任务 A 从被抢占处继续执行。其修改了一个 PORTA 的拷贝，这其实只是寄存器在任务 A 回写到 PORTA 之前曾经保存过的值。

任务 A 更新并回写了一个过期的 PORTA 寄存器值。在任务 A 获得拷贝与更新回写之间，任务 B 又修改了 PORTA 的值。而之后任务 A 对 PORTA 的回写操作，覆盖了任务 B 对 PORTA 进行的修改结果，效果上等同于破坏了 PORTA 寄存器的值。

虽然是以一个外围设备寄存器为例，但是整个情形同样适用于全局变量的读-改-写操作。

3. 变量的非原子访问

更新结构体的多个成员变量，或是更新的变量其长度超过了架构体系的自然长度(比如，更新一个 16 位机上的 32 位变量)均是非原子操作的例子。如果这样的操作被中断，将可能导致数据损坏或丢失。

4. 函数重入

如果一个函数可以安全地被多个任务调用，或是在任务与中断中均可调用，则这个函数是可重入的。

每个任务都单独维护自己的栈空间及其自身在的内存寄存器组中的值。**如果一个函数除了访问自己栈空间上分配的数据或是内核寄存器中的数据外，不会访问其它任何数据**，则这个函数就是可重入的。程序清单 58 示例了一个可重入函数，而程序清单 59 是一个不可重入函数的例子。

```
/* A parameter is passed into the function. This will either be
passed on the stack or in a CPU register. Either way is safe as
each task maintains its own stack and its own set of register
values. */
long lAddOneHundered( long lVar1 )
{
/* This function scope variable will also be allocated to the stack
or a register, depending on compiler and optimization level. Each
task or interrupt that calls this function will have its own copy
of lVar2. */
long lVar2;
    lVar2 = lVar1 + 100;

    /* Most likely the return value will be placed in a CPU register,
    although it too could be placed on the stack. */
    return lVar2;
}
```

程序清单 **58** 可重入函数示例

```
/* In this case lVar1 is a global variable so every task that calls
the function will be accessing the same single copy of the variable. */
long lVar1;
long lNonsenseFunction( void )
{
/* This variable is static so is not allocated on the stack. Each task
that calls the function will be accessing the same single copy of the
variable. */
static long lState = 0;
long lReturn;

    switch( lState )
    {
        case 0 : lReturn = lVar1 + 10;
                 lState = 1;
                 break;

        case 1 : lReturn = lVar1 + 20;
                 lState = 0;
                 break;
    }
}
```

程序清单 **59** 不可重入函数示例

互斥

访问一个被多任务共享，或是被任务与中断共享的资源时，需要采用"互斥"技术以保证数据在任何时候都保持一致性。这样做的目的是要确保任务从开始访问资源就具有排它性，直至这个资源又恢复到完整状态。

FreeRTOS 提供了多种特性用以实现互斥，但是最好的互斥方法（如果可能的话，任何时候都当如此）还是通过精心设计应用程序，尽量不要共享资源，或者是每个资源都通过单任务访问。

本章期望让读者了解以下内容：

- 为什么，以及在什么时候有必要进行资源管理与控制。
- 什么是临界区。
- 互斥是什么意思。
- 挂起调度器有什么意义。
- 如何使用互斥量。
- 如何创建与使用守护任务。
- 什么是优先级反转，以及优先级继承是如何减小(但不是消除)其影响的。

4.2 临界区与挂起调度器

基本临界区

基本临界区是指宏 taskENTER_CRITICAL()与 taskEXIT_CRITICAL()之间的代码区间，**程序清单 60** 是一段范例代码。Critical Sections 也被称作 Critical Regions。

```
/* 为了保证对PORTA寄存器的访问不被中断，将访问操作放入临界区。
 进入临界区 */
taskENTER_CRITICAL();

/* 在taskENTER_CRITICAL() 与 taskEXIT_CRITICAL()之间不会切换到其它任务。 中断可以执行，也允许
嵌套，但只是针对优先级高于configMAX_SYSCALL_INTERRUPT_PRIORITY的中断 – 而且这些中断不允许访问
FreeRTOS API 函数. */
PORTA |= 0x01;

/* 我们已经完成了对PORTA的访问，因此可以安全地离开临界区了。 */
taskEXIT_CRITICAL();
```

程序清单 60 使用临界区对寄存器的访问进行保护

本书采用的范例工程使用了一个名为 vPrintString()的函数，用以往标准输出设备写字符串。这个标准输出即是 Open Watcom DOS 可执行程序的终端窗口。vPrintString()被多个不同的任务调用，所以理论上其函数实现中应当使用一个临界区对标准输出进行保护。如**程序清单 61** 所示。

```
void vPrintString( const portCHAR *pcString )
{
    /* 往stdout中写字符串，使用临界区这种原始的方法实现互斥。 */
    taskENTER_CRITICAL();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    taskEXIT_CRITICAL();

    /* 允许按任意键停止应用程序运行。实际的应用程序如果有使用到键值，还需要对键盘输入进行保护。 */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

程序清单 61 vPrintString()的一种可能的实现方法

临界区是提供互斥功能的一种非常原始的实现方法。临界区的工作仅仅是简单地把中断全部关掉，或是关掉优先级在 configMAX_SYSCAL_INTERRUPT_PRIORITY 及以下的中断——依赖于具体使用的 FreeRTOS 移植。抢占式上下文切换只可能在某个中断中完成，所以调用 taskENTER_CRITICAL()的任务可以在中断关闭的时段一直保持运行态，直到退出临界区。

临界区必须只具有很短的时间，否则会反过来影响中断响应时间。在每次调用 taskENTER_CRITICAL()之后，必须尽快地配套调用一个 taskEXIT_CRITICAL()。从这个角度来看，对标准输出的保护不应当采用临界区(如**程序清单 61** 所示)，因为写终端在时间上会是一个相对较长的操作。DOS 模拟器和 Open Watcom 处理终端输出时没有采用这种互斥方式，其库函数中是没有关中断的。本章中的示例代码会探索其它解决方案。

临界区嵌套是安全的，因为内核有维护一个嵌套深度计数。临界区只会在嵌套深度为 0 时才会真正退出——即在为每个之前调用的 taskENTER_CRITICAL()都配套调用了 taskEXIT_CRITICAL()之后。

挂起(锁定)调度器

也可以通过挂起调度器来创建临界区。挂起调度器有些时候也被称为锁定调度器。

基本临界区保护一段代码区间不被其它任务或中断打断。由挂起调度器实现的临界区只可以保护一段代码区间不被其它任务打断，因为这种方式下，中断是使能的。

如果一个临界区太长而不适合简单地关中断来实现，可以考虑采用挂起调度器的方式。但是唤醒(resuming, or un-suspending)调度器却是一个相对较长的操作。所以评估哪种是最佳方式需要结合实际情况。

vTaskSuspendAll() API 函数

```
void vTaskSuspendAll( void );
```

程序清单 62 vTaskSuspendAll() API 函数原型

通过调用 vTaskSuspendAll()来挂起调度器。挂起调度器可以停止上下文切换而不用关中断。如果某个中断在调度器挂起过程中要求进行上下文切换，则个这请求也会被挂起，直到调度器被唤醒后才会得到执行。

在调度器处于挂起状态时，不能调用 FreeRTOS API 函数。

xTaskResumeAll() API 函数

```
portBASE_TYPE xTaskResumeAll( void );
```

程序清单 63 xTaskResumeAll() API 函数原型

表 18 xTaskResumeAll()返回值

参数名	描述
返回值	在调度器挂起过程中，上下文切换请求也会被挂起，直到调度器被唤醒后才会得到执行。如果一个挂起的上下文切换请求在 xTaskResumeAll()返回前得到执行，则函数返回 pdTRUE。在其它情况下，xTaskResumeAll()返回 pdFALSE。

嵌套调用 vTaskSuspendAll()和 xTaskResumeAll()是安全的，因为内核有维护一个嵌套深度计数。调度器只会在嵌套深度计数为 0 时才会被唤醒——即在为每个之前调用的 vTaskSuspendAll()都配套调用了 xTaskResumAll()之后。

程序清单 64 展示了实际使用的 vPrintString()实现代码。这种实现方式即是通过挂起调度器的方式来保护终端输出。

```
void vPrintString( const portCHAR *pcString )
{
    /* Write the string to stdout, suspending the scheduler as a method
    of mutual exclusion. */
    vTaskSuspendScheduler();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    xTaskResumeScheduler();

    /* Allow any key to stop the application running. A real application that
    actually used the key value should protect access to the keyboard input too. */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

程序清单 **64 vPrintString()**实现代码

4.3 互斥量(及二值信号量)

互斥量是一种特殊的二值信号量，用于控制在两个或多个任务间访问共享资源。单词MUTEX(互斥量)源于"MUTual EXclusion"。

在用于互斥的场合，互斥量从概念上可看作是与共享资源关联的令牌。一个任务想要合法地访问资源，其必须先成功地得到(Take)该资源对应的令牌(成为令牌持有者)。当令牌持有者完成资源使用，其必须马上归还(Give)令牌。只有归还了令牌，其它任务才可能成功持有，也才可能安全地访问该共享资源。一个任务除非持有了令牌，否则不允许访问共享资源。这种机制在**图 36** 中展示。

虽然互斥量与二值信号量之间具有很多相同的特性，但**图 36** 展示的情形(互斥量用于互斥功能)完全不同于**图 30** 展示的情形(二值信号量用于同步)。两者间最大的区别在于信号量在被获得之后所发生的事情：

- 用于互斥的信号量必须归还。
- 用于同步的信号量通常是完成同步之后便丢弃，不再归还。

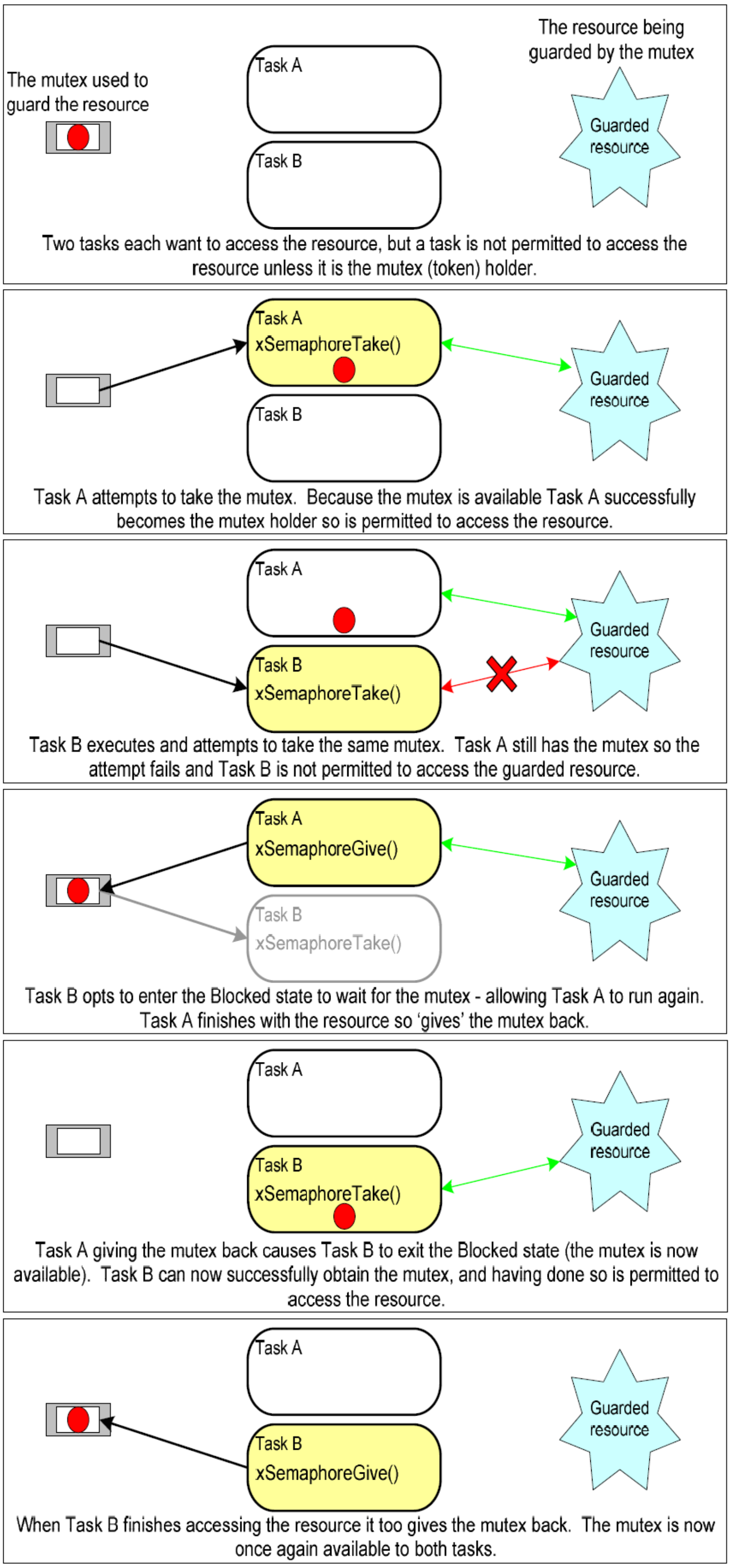


图 **36** 互斥量用于互斥功能

这种机制纯粹是工作于应用程序作者制定的规则之下。任务不是在任何时候都可以访问资源是不需要理由的，因为这是所有任务达成的一致，除非它们能成为互斥量的持有者。

xSemaphoreCreateMutex() API 函数

互斥量是一种信号量。FreeRTOS 中所有种类的信号量句柄都保存在类型为 xSemaphoreHandle 的变量中。

互斥量在使用前必须先创建。创建一个互斥量类型的信号量需要使用 xSemaphoreCreateMutex() API 函数。

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

程序清单 65　xSemaphoreCreateMutex() API 函数原型

表 19　xSemaphoreCreateMutex()返回值

参数名	描述
返回值	如果返回 NULL 表示互斥量创建失败。原因是内存堆空间不足导致 FreeRTOS 无法为互斥量分配结构数据空间。第五章提供更多关于内存管理方面的信息。 返回非 NULL 值表示互斥量创建成功。返回值应当保存起来作为该互斥量的句柄。

例 15. 使用信号量重写 vPrintString()

本例创建了一个新版本的 vPrintString()，称为 prvNewPrintString()，然后在多任务中调用这个新版函数。prvNewPrintString()具有与 vPrintString()完全相同的功能，只是在实现上使用互斥量代替基本临界区来实现对标准输出的控制。prvNewPrintString()的实现代码参见**程序清单 66**。

```
static void prvNewPrintString( const portCHAR *pcString )
{
    /* 互斥量在调度器启动之前就已创建，所以在此任务运行时信号量就已经存在了。

    试图获得互斥量。如果互斥量无效，则将阻塞，进入无超时等待。xSemaphoreTake()只可能在成功获得互
    斥量后返回，所以无需检测返回值。如果指定了等待超时时间，则代码必须检测到xSemaphoreTake()返回
    pdTRUE后，才能访问共享资源(此处是指标准输出)。 */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* 程序执行到这里表示已经成功持有互斥量。现在可以自由访问标准输出，因为任意时刻只会有一个任
        务能持有互斥量。 */
        printf( "%s", pcString );
        fflush( stdout );

        /* 互斥量必须归还！ */
    }
    xSemaphoreGive( xMutex );

    /* Allow any key to stop the application running. A real application that
    actually used the key value should protect access to the keyboard too. A
    real application is very unlikely to have more than one task processing
    key presses though! */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

程序清单 66 **prvNewPrintString()**实现代码

prvNewPrintString()被一个任务的两个实例重复调用。在每次调用之间采用了一个随机延迟时间。任务的入口参数用于向任务的每个实例传递各自的输出字符串。任务prvPrintTask()的实现代码参见**程序清单 67**。

```
static void prvPrintTask( void *pvParameters )
{
char *pcStringToPrint;

    /* Two instances of this task are created so the string the task will send
    to prvNewPrintString() is passed into the task using the task parameter.
    Cast this to the required type. */
    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        /* Print out the string using the newly defined function. */
        prvNewPrintString( pcStringToPrint );

        /* 等待一个伪随机时间。注意函数rand()不要求可重入，因为在本例中rand()的返回值并不重要。但
        在安全性要求更高的应用程序中，需要用一个可重入版本的rand()函数 - 或是在临界区中调用rand()
        函数。 */
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}
```

程序清单 67　例 15 中任务 prvPrintTask() 的实现代码

和往常一样，main()函数简单地创建互斥量，创建任务，然后启动调度器。具体实现代码参见**程序清单 68**。

任务 prvPintTask()的两个实例在创建时指定了不同的优先级。所以运行时，低优先级任务在有些时候会被高优先级的任务抢占。由于使用了互斥量来保证每个任务在访问终端时保持互斥，所以即使是任务被抢占，字符串也会正确显示，而不会有其它导致破坏的可能。任务的抢占频率还可以再提高，只需要减少任务在阻塞态中花费的最长时间，本例中这个最长时间默认为 0x1ff 个系统心跳周期。

```
int main( void )
{
    /* 信号量使用前必须先创建。本例创建了一个互斥量类型的信号量。 */
    xMutex = xSemaphoreCreateMutex();

    /* 本例中的任务会使用一个随机延迟时间，这里给随机数发生器生成种子。 */
    srand( 567 );

    /* Check the semaphore was created successfully before creating the tasks. */
    if( xMutex != NULL )
    {
        /* Create two instances of the tasks that write to stdout. The string
        they write is passed in as the task parameter. The tasks are created
        at different priorities so some pre-emption will occur. */
        xTaskCreate( prvPrintTask, "Print1", 1000,
                    "Task 1 ******************************************\r\n", 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 1000,
                    "Task 2 ------------------------------------------\r\n", 2, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }

    /* 如果一切正常，main()函数不会执行到这里，因为调度器已经开始运行任务。但如果程序运行到了这里，
    很可能是由于系统内存不足而无法创建空闲任务。第五章会提供更多关于内存管理的信息 */
    for( ;; );
}
```

程序清单 68 例 15 的 main()函数实现

图 37 展示了例 15 的运行输出结果。一种可能的执行流程参见图 38。

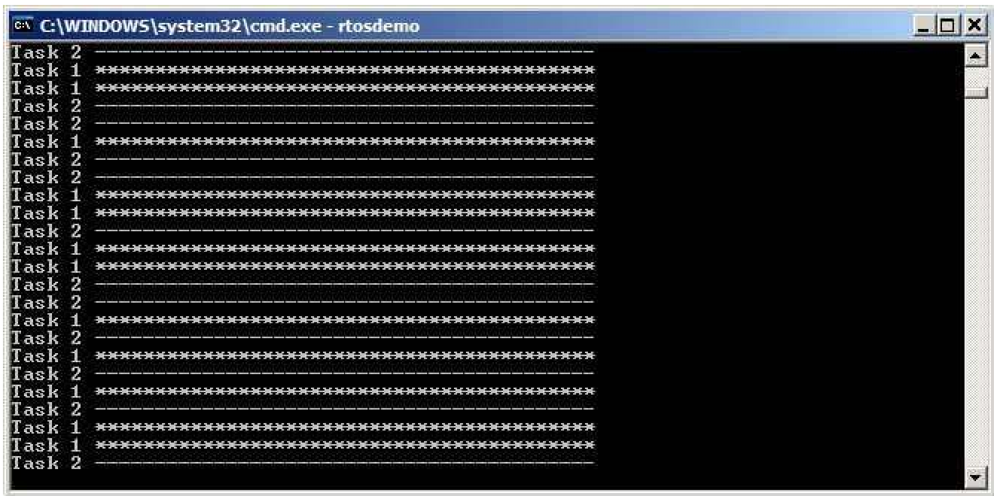


图 37 例 15 的输出结果

从图 37 中可以看到，和所期望的一样，终端上显示的字符串没有遭到破坏。随机的显示顺序是任务采用随机延迟周期的结果。

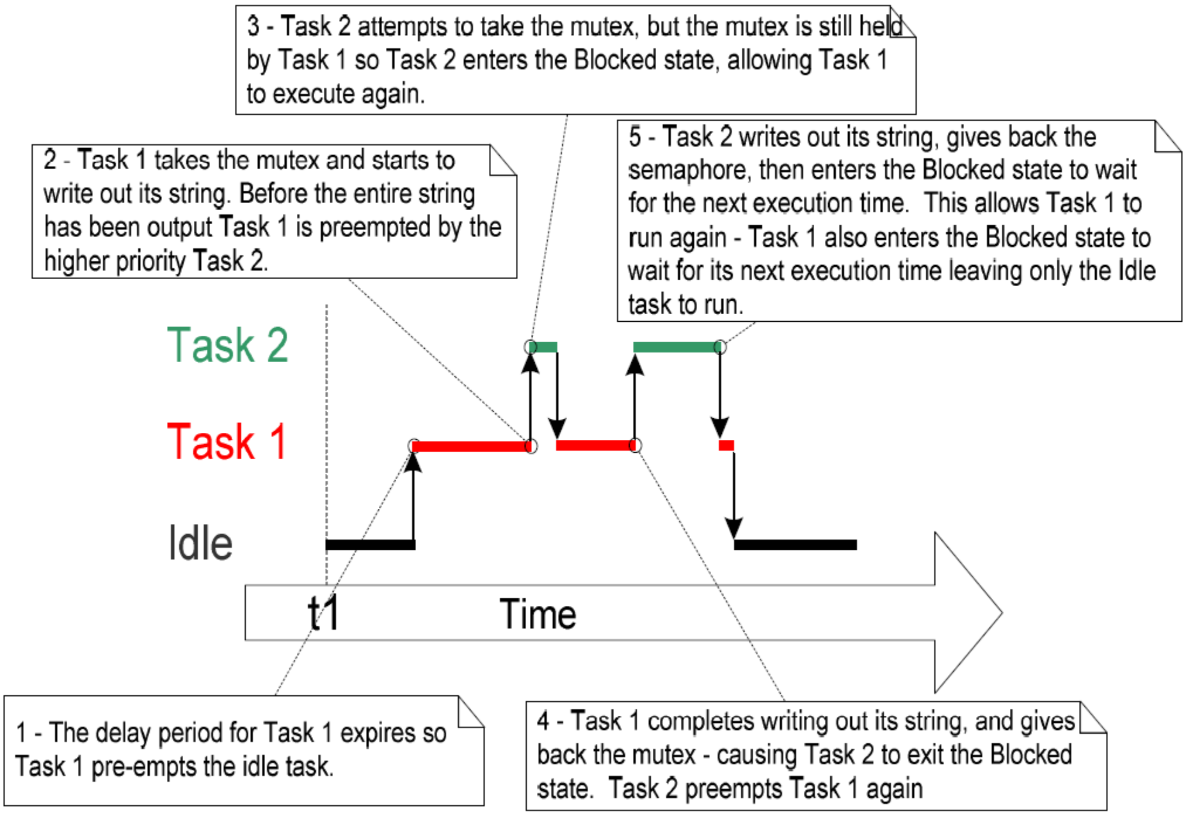


图 38　例 15 中一种可能的执行流程

优先级反转

图 38 也展现出了采用互斥量提供互斥功能的潜在缺陷之一。在这种可能的执行流程描述中，高优先级的任务 2 竟然必须等待低优先级的任务 1 放弃对互斥量的持有权。高优先级任务被低优先级任务阻塞推迟的行为被称为"优先级反转"。这是一种不合理的行为方式，如果把这种行为再进一步放大，当高优先级任务正等待信号量的时候，一个介于两个任务优先之间的中等优先级任务开始执行——这就会导致一个高优先级任务在等待一个低优先级任务，而低优先级任务却无法执行！这种最坏的情形在图 39 中进行展示。

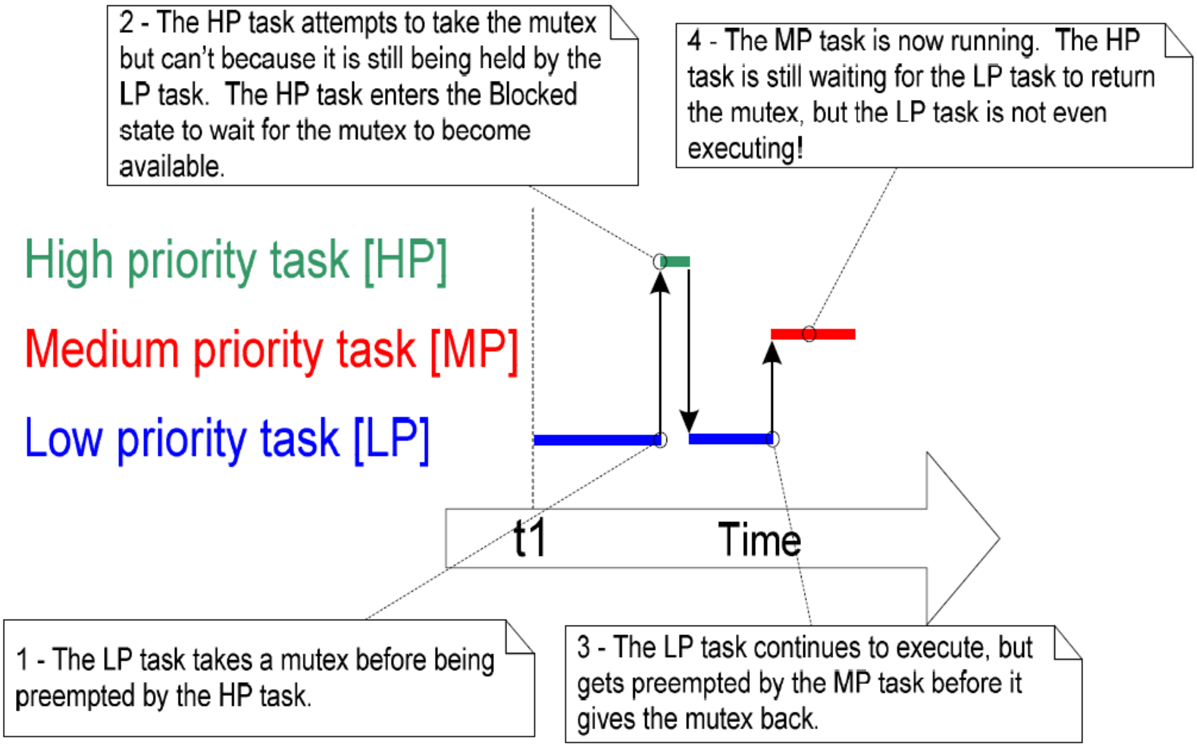


图 **40** 优先级反转的一种最坏情况

优先级反转可能会产生重大问题。但是在一个小型的嵌入式系统中，通常可以在设计阶段就通过规划好资源的访问方式避免出现这个问题。

优先级继承

FreeRTOS 中互斥量与二值信号量十分相似——唯一的区别就是互斥量自动提供了一个基本的"优先级继承"机制。优先级继承是最小化优先级反转负面影响的一种方案——其并不能修正优先级反转带来的问题，仅仅是减小优先级反转的影响。优先级继承使得系统行为的数学分析更为复杂，所以如果可以避免的话，并不建议系统实现对优先级继承有所依赖。

优先级继承暂时地将互斥量持有者的优先级提升至所有等待此互斥量的任务所具有的最高优先级。持有互斥量的低优先级任务"继承"了等待互斥量的任务的优先级。这种机制在图 **40** 中进行展示。互斥量持有者在归还互斥量时，优先级会自动设置为其原来的优先级。

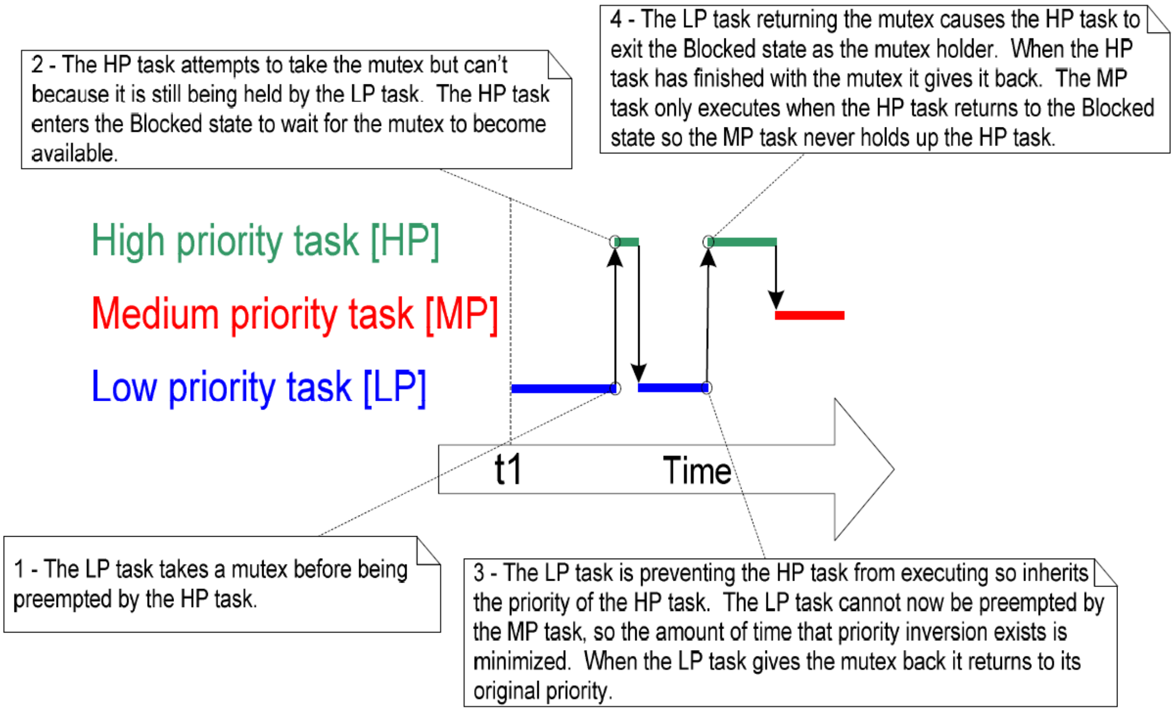


图 40　优先级继承最小化优先级反转的影响

由于最好是优先考虑避免优先级反转，并且因为 FreeRTOS 本身是面向内存有限的微控制器，所以只实现了最基本的互斥量的优先级继承机制，这种实现假定一个任务在任意时刻只会持有一个互斥量。

死锁

死锁是利用互斥量提供互斥功能的另一个潜在缺陷。Deadlock 有时候会被更戏剧性地称为"deadly embrace(抱死)"。

当两个任务都在等待被对方持有的资源时，两个任务都无法再继续执行，这种情况就被称为死锁。考虑如下情形，任务 A 与任务 B 都需要获得互斥量 X 与互斥量 Y 以完成各自的工作：

1. 任务 A 执行，并成功获得了互斥量 X。
2. 任务 A 被任务 B 抢占。
3. 任务 B 成功获得了互斥量 Y，之后又试图获取互斥量 X——但互斥量 X 已经被任务 A 持有，所以对任务 B 无效。任务 B 选择进入阻塞态以等待互斥量 X 被释放。
4. 任务 A 得以继续执行。其试图获取互斥量 Y——但互斥量 Y 已经被任务 B 持有而对任务 A 无效。任务 A 也选择进入阻塞态以等待互斥量 Y 被释放。

这种情形的最终结局是，任务 A 在等待一个被任务 B 持有的互斥量，而任务 B 也在等待一个被任务 A 持有的互斥量。死锁于是发生，因为两个任务都不可能再执行下去了。

和优先级反转一样，避免死锁的最好方法就是在设计阶段就考虑到这种潜在风险，这样设计出来的系统就不应该会出现死锁的情况。于实践经验而言，对于一个小型嵌入式系统，死锁并不是一个大问题，因为系统设计者对整个应用程序都非常清楚，所以能够找出发生死锁的代码区域，并消除死锁问题。

4.4 守护任务

守护任务提供了一种干净利落的方法来实现互斥功能，而不用担心会发生优先级反转和死锁。

守护任务是对某个资源具有唯一所有权的任务。只有守护任务才可以直接访问其守护的资源——其它任务要访问该资源只能间接地通过守护任务提供的服务。

例 16. 采用守护任务重写 vPrintString()

例 16 提供了 vPrintString()的另一种实现方法，这里采用了一个守护任务来管理对标准输出的访问。当一个任务想要往终端写信息的时候，其不能直接调用打印函数，而是将消息发送到守护任务。

守护任务使用了一个 FreeRTOS 队列来对终端实现串行化访问。该任务内部实现不必考虑互斥，因为它是唯一能够直接访问终端的任务。

守护任务大部份时间都在阻塞态等待队列中有信息到来。当一个信息到达时，守护任务仅仅简单地将收到的信息写到标准输出上，然后又返回阻塞态，继续等待下一条信息地到来。守护任务的具体实现参见程序**清单 70**。

中断中可以写队列，所以中断服务例程也可以安全地使用守护任务提供的服务，从而把信息输出到终端。在本例中，一个心跳中断钩子函数用于每 200 心跳周期就输出一个消息。

心跳钩子函数(或称回调函数)由内核在每次心跳中断时调用。要挂接一个心跳钩子函数，需要做以下配置：

- 设置 FreeRTOSConfig.h 中的常量 configUSE_TICK_HOOK 为 1。
- 提供钩子函数的具体实现，要求使用程序**清单 69** 中的函数名和原型。

```
void vApplicationTickHook( void );
```

程序清单 69 心跳钩子函数名及原型

心跳钩子函数在系统心跳中断的上下文上执行，所以必须保证非常短小，适度占用栈空间，并且不要调用任何名字不带后缀”FromISR”的 FreeRTOS API 函数。

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
char *pcMessageToPrint;
    /* 这是唯一允许直接访问终端输出的任务。任何其它任务想要输出字符串，都不能直接访问终端，而是将要
    输出的字符串发送到此任务。并且因为只有本任务才可以访问标准输出，所以本任务在实现上不需要考虑互斥
    和串行化等问题。 */
    for( ;; )
    {
        /* 等待信息到达。指定了一个无限长阻塞超时时间，所以不需要检查返回值 – 此函数只会在成功收到
        消息时才会返回。 */
        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

        /* 输出收到的字符串。 */
        printf( "%s", pcMessageToPrint );
        fflush( stdout );

        /* Now simply go back to wait for the next message. */
    }
}
```

程序清单 **70** 守护任务

信息输出任务与**例 15** 相似，不同的是本例中字符串是通过队列发送到守护任务，而不是直接输出到终端。具体实现参见**程序清单 71**。和之前一样，为这个任务创建了两个独立的实例，每个实例输出各自从任务入口参数传入的字符串。

```
static void prvPrintTask( void *pvParameters )
{
int iIndexToString;
    /* Two instances of this task are created. The task parameter is used to pass
    an index into an array of strings into the task. Cast this to the required type. */
    iIndexToString = ( int ) pvParameters;
    for( ;; )
    {
        /* 打印输出字符串，不能直接输出，通过队列将字符串指针发送到守护任务。队列在调度器启动之前就
        创建了，所以任务执行时队列就已经存在了。并有指定超时等待时间，因为队列空间总是有效。 */
        xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ] ), 0 );

        /* 等待一个伪随机时间。注意函数rand()不要求可重入，因为在本例中rand()的返回值并不重要。但
        在安全性要求更高的应用程序中，需要用一个可重入版本的rand()函数 – 或是在临界区中调用rand()
        函数。 */
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}
```

程序清单 **71** 例 **16** 中的打印输出任务实现代码

心跳钩子函数仅仅是简单地对其被调用次数进行计数，当计数至 200 时就向守护任务发送信息。为了具有更好的演示效果，心跳钩子函数将信息发送到队列首，而打印输出任务将信息发送到队列尾。心跳钩子函数的实现代码如**程序清单 72** 所示。

```
void vApplicationTickHook( void )
{
static int iCount = 0;
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* Print out a message every 200 ticks. The message is not written out
    directly, but sent to the gatekeeper task. */
    iCount++;
    if( iCount >= 200 )
    {
        /* In this case the last parameter (xHigherPriorityTaskWoken) is not
        actually used but must still be supplied. */
        xQueueSendToFrontFromISR( xPrintQueue,
                                  &( pcStringsToPrint[ 2 ] ),
                                  &xHigherPriorityTaskWoken );

        /* Reset the count ready to print out the string again in 200 ticks
        time. */
        iCount = 0;
    }
}
```

程序清单 72　心跳钩子函数实现代码

和往常一样，main()函数创建队列和所有任务，然后启动调度器。main()函数的具体实现参见**程序清单 73**。

```
/* 定义任务和中断将会通过守护任务输出的字符串。 */
static char *pcStringsToPrint[] =
{
    "Task 1 ****************************************************\r\n",
    "Task 2 ----------------------------------------------------\r\n",
    "Message printed from the tick hook interrupt ##############\r\n"
};

/*-----------------------------------------------------------*/
/* 声明xQueueHandle变量。这个变量将会用于打印任务和中断往守护任务发送消息。*/
xQueueHandle xPrintQueue;
/*-----------------------------------------------------------*/

int main( void )
{
    /* 创建队列，深度为5，数据单元类型为字符指针。 */
    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

    /* 为伪随机数发生器产生种子。 */
    srand( 567 );

    /* Check the queue was created successfully. */
    if( xPrintQueue != NULL )
    {
        /* 创建任务的两个实例，用于向守护任务发送信息。任务入口参数传入需要输出的字符串索引号。这两
        个任务具有不同的优先级，所以高优先级任务有时会抢占低优先级任务。 */
        xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

        /* 创建守护任务。这是唯一一个允许直接访问标准输出的任务。 */
        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }

    /* 如果一切正常，main()函数不会执行到这里，因为调度器已经开始运行任务。但如果程序运行到了这里，
    很可能是由于系统内存不足而无法创建空闲任务。第五章会提供更多关于内存管理的信息 */
    for( ;; );
}
```

程序清单 **73** 例 **16** 中的 **main()**函数实现代码

图 41 展现了**例 16** 的运行输出结果。从图中可以看到，无念经是来自任务的字符串还是来自中断的字符串都被正确地打印输出，没有出现数据破坏。

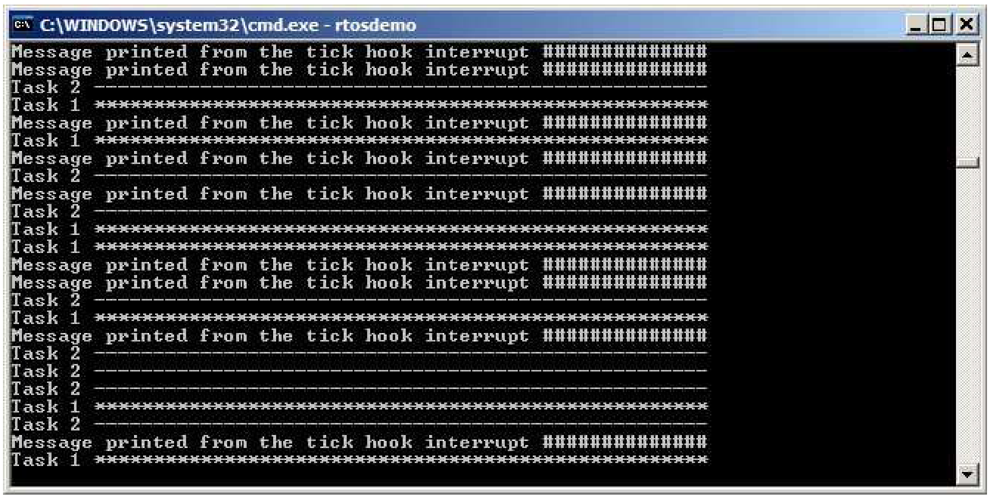


图 41 例 16 的运行输出结果

守护任务的优先级低于打印任务——所以发送到守护任务的消息会一直保持在队列中，直到两个打印任务都进入阻塞态。在一些情况下，需要给守护任务赋予一个较高的优先级，消息就可以得到更快的处理——但这样做会由于守护任务的开销使得低优先级任务被推迟，直到守护任务完成对受其保护的资源的访问。

第五章

内存管理

5.1 概览

每当任务，队列或是信号量被创建时，内核需要进行动态内存分配。虽然可以调用标准的 malloc()与 free()库函数，但必须承担以下若干问题：

1. 这两个函数在小型嵌入式系统中可能不可用。
2. 这两个函数的具体实现可能会相对较大，会占用较多宝贵的代码空间。
3. 这两个函数通常不具备线程安全特性。
4. 这两个函数具有不确定性。每次调用时的时间开销都可能不同。
5. 这两个函数会产生内存碎片。
6. 这两个函数会使得链接器配置得复杂。

不同的嵌入式系统具有不同的内存配置和时间要求。所以单一的内存分配算法只可能适合部分应用程序。因此，FreeRTOS 将内存分配作为可移植层面(相对于基本的内核代码部分而言)。这使得不同的应用程序可以提供适合自身的具体实现。

当内核请求内存时，其调用 pvPortMalloc()而不是直接调用 malloc()；当释放内存时，调用 vPortFree()而不是直接调用 free()。pvPortMalloc()具有与 malloc()相同的函数原型；vPortFree()也具有与 free()相同的函数原型。

FreeRTOS 自带有三种 pvPortMalloc()与 vPortFree()实现范例，这三种方式都会在本章描述。FreeRTOS 的用户可以选用其中一种，也可以采用自己的内存管理方式。

这三个范例对应三个源文件：heap_1.c，heap_2.c，heap_3.c——这三个文件都放在目录 FreeRTOS\Source\Portable\MemMang 中。早期版本的 FreeRTOS 所采用的原始内存池和内存块分配方案已经被移除了，因为定义内存块和内存池的大小需要较深入的努力和理解。

在小型嵌入式系统中，通常是在启动调度器之前创建任务，队列和信号量。这种情况表明，动态分配内存只会出现在应用程序真正开始执行实时功能之前，而且内存一旦分配就不会再释放。这就意味着选择内存分配方案时不必考虑一些复杂的因素，比如确定性与内存碎片等，而只需要从性能上考虑，比如代码大小和简易性。

本章期望让读者了解以下事情：

- FreeRTOS 在什么时候分配内存。
- FreeRTOS 提供的三种内存分配方案范例。

5.2 内存分配方案范例

Heap_1.c

Heap_1.c 实现了一个非常基本的 pvPortMalloc()版本，而且没有实现 vPortFree()。如果应用程序不需要删除任务，队列或者信号量，则具有使用 heap_1 的潜质。Heap_1 总是具有确定性。

这种分配方案是将 FreeRTOS 的内存堆空间看作一个简单的数组。当调用 pvPortMalloc()时，则将数组又简单地细分为更小的内存块。

数组的总大小(字节为单位)在 FreeRTOSConfig.h 中由 configTOTAL_HEAP_SIZE 定义。以这种方式定义一个巨型数组会让整个应用程序看起来耗费了许多内存——即使是在数组没有进行任何实际分配之前。

需要为每个创建的任务在堆空间上分配一个任务控制块(TCB)和一个栈空间。图 **42** 展示了 heap_1 是如何在任务创建时细分这个简单数组的。从图 **42** 中可以看到：

- A 表示数组在没有任何任务创建时的情形，这里整个数据是空的。
- B 表示数组在创建了一个任务后的情形。
- C 表示数组在创建了三个任务后的情形。

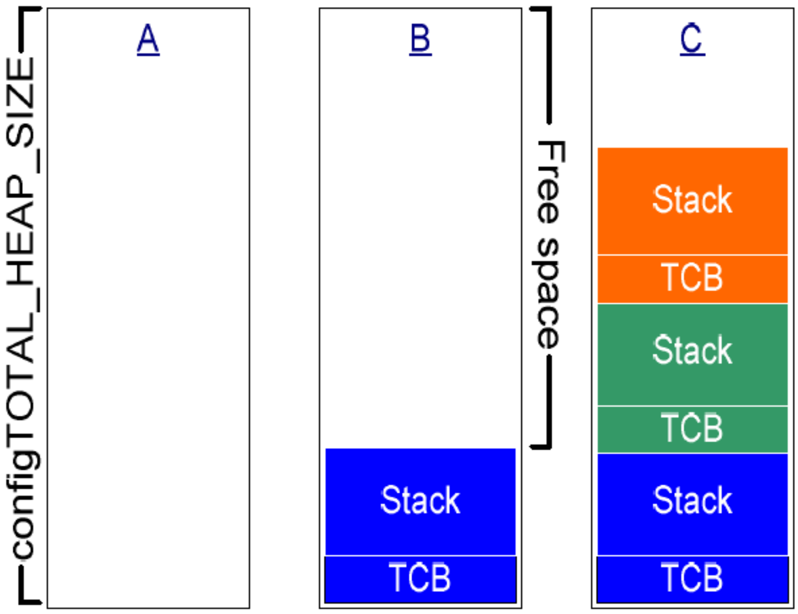


图 **42**　每次创建任务后的内存分配情况

Heap_2.c

Heap_2.c 也是使用了一个由 configTOTAL_HEAP_SIZE 定义大小的简单数组。不同于 heap_1 的是，heap_2 采用了一个最佳匹配算法来分配内存，并且支持内存释放。由于声明了一个静态数组，所以会让整个应用程序看起来耗费了许多内存——即使是在数组没有进行任何实际分配之前。

最佳匹配算法保证 pvPortMalloc()会使用最接近请求大小的空闲内存块。比如，考虑以下情形：

1. 堆空间中包含了三个空闲内存块，分别为 5 字节，25 字节和 100 字节大小。
2. pvPortMalloc()被调用以请求分配 20 字节大小的内存空间。

匹配请求字节数的最小空闲内存块是具有25字节大小的内存块——所以pvPortMalloc()会将这个 25 字节块再分为一个 20 字节块和一个 5 字节块 [3]，然后返回一个指向 20 字节块的指针。剩下的 5 字节块则保留下来，留待以后调用 pvPortMalloc()时使用。

Heap_2.c 并不会把相邻的空闲块合并成一个更大的内存块，所以会产生内存碎片——如果分配和释放的总是相同大小的内存块，则内存碎片就不会成为一个问题。Heap_2.c 适合用于那些重复创建与删除具有相同栈空间任务的应用程序。

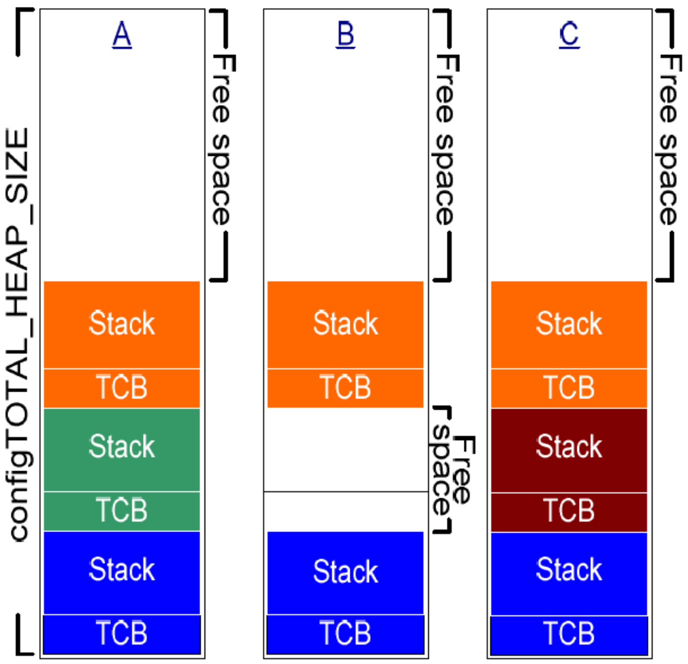


图 43 创建和删除任务后的内存分配情况

[3] 这是过于简化。因为 heap_2 会在堆中保存一些信息，所以两个分离的内存块总量会小于 25

图 **43** 展示了当任务创建，删除以及再创建过程中，最佳匹配算法是如何工作的。从图 **43** 中可以看出：

- A 表示数组在创建了三个任务后的情形。数组的顶部还剩余一个大空闲块。
- B 表示数组在删除了一个任务后的情形。顶部的大空闲块保持不变，并多出了两个小的空闲块，分别是被删除任务的 TCB 和任务栈。
- C 表示数组在又创建了一个任务后的情形。创建一个任务会产生两次调用 pvPortMalloc()，一次是分配 TCB，一次是分配任务栈(调用 pvPortMalloc()发生在 xTaskCreate() API 函数内部)。

  每个 TCB 都具有相同大小，所以最佳匹配算法可以确保之前被删除的任务占用的 TCB 空间被重新分配用作新任务的 TCB 空间。

  新建任务的栈空间与之前被删除任务的栈空间大小相同，所以最佳匹配算法会保证之前被删除任务占用的栈空间会被重新分配用作新任务的栈空间。

  数组顶部的大空闲块依然保持不变。

Heap_2.c 虽然不具备确定性，但是比大多数标准库实现的 malloc()与 free()更有效率。

Heap_3.c

Heap_3.c 简单地调用了标准库函数 malloc()和 free()，但是通过暂时挂起调度器使得函数调用备线程安全特性。其实现代码参见**程序清单 74**。

此时的内存堆空间大小不受 configTOTAL_HEAP_SIZE 影响，而是由链接器配置决定。

```
void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn;
    vTaskSuspendAll();
    {
        pvReturn = malloc( xWantedSize );
    }
    xTaskResumeAll();
    return pvReturn;
}


void vPortFree( void *pv )
{
    if( pv != NULL )
    {
        vTaskSuspendAll();
        {
            free( pv );
        }
        xTaskResumeAll();
    }
}
```

程序清单 74 **heap_3.c** 实现代码

第六章

错误排查

6.1 概览

本章主要是为刚接触 FreeRTOS 的用户指出那些新手通常容易遇到的问题。这里把最主要的篇幅放在栈溢出以及栈溢出侦测上，因为栈相关的问题是过去几年遇到最多的问题。对其它一些比较常见的问题，本章简要的以 FAQ(问答)的形式给出可能的原因和解决方法。

printf-stdarg.c

当调用标准 C 库函数时，栈空间使用量可能会急剧上升，特别是 IO 与字符串处理函数，比如 sprintf()。在 FreeRTOS 下载包中有一个名为 printf-stdarg.c 的文件。这个文件实现了一个栈效率优化版的小型 sprintf()，可以用来代替标准 C 库函数版本。在大多数情况下，这样做可以使得调用 sprintf()及相关函数的任务对栈空间的需求量小很多。

printf-stdarg.c 源代码开放，但是为第三方所有。所以此源代码的 license 独立于 FreeRTOS。具体的 license 条款包含在该源文件的起始部分。

6.2 栈溢出

FreeRTOS 提供了多种特性来辅助跟踪调试栈相关的问题[4]。

uxTaskGetStackHighWaterMark() API 函数

每个任务都独立维护自己的栈空间，栈空间总量在任务创建时进行设定。uxTaskGetStackHighWaterMark()主要用来查询指定任务的运行历史中，其栈空间还差多少就要溢出。这个值被称为栈空间的”高水线(High Water Mark)”。

```
unsigned portBASE_TYPE uxTaskGetStackHighWaterMark( xTaskHandle xTask );
```

程序清单 75 uxTaskGetStackHighWaterMark() API 函数原型

表 20 uxTaskGetStackHighWaterMark()参数与返回值

参数名	描述
xTask	被查询任务的句柄——欲知如何获得任务句柄，详情请参见 API 函数 xTaskCreate()的参数 pxCreatedTask。 如果传入 NULL 句柄，则任务查询的是自身栈空间的高水线。
返回值	任务栈空间的实际使用量会随着任务执行和中断处理过程上下浮动。uxTaskGetStackHighWaterMark()返回从任务启动执行开始的运行历史中，栈空间具有的最小剩余量。这个值即是栈空间使用达到最深时的剩下的未使用的栈空间。这个值越是接近 0，则这个任务就越是离栈溢出不远了。

[4] 不幸的是，这些特性无法在一个模拟的 DOS 环境下使用，因为 DOS 采用的是段式内存。因此无法在 Open Watcom 上的示范这些特性的使用方法。

运行时栈侦测 —— 概述

FreeRTOS 包含两种运行时栈侦测机制，由 FreeRTOSConfig.h 中的配置常量 configCHECK_FOR_STACK_OVERFLOW 进行控制。这两种方式都会增加上下切换开销。

栈溢出钩子函数(或称回调函数)由内核在侦测到栈溢出时调用。要使用栈溢出钩子函数，需要进行以下配置：

- 在 FreeRTOSConfig.h 中把 configCHECK_FOR_STACK_OVERFLOW 设为 1 或 2。
- 提供钩子函数的具体实现，采用**程序清单 76** 所示的函数名和函数原型。

```
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR *pcTaskName );
```

程序清单 76 栈溢出钩子函数原型

栈溢出钩子函数只是为了使跟踪调试栈空间错误更容易，而无法在栈溢出时对其进行恢复。函数的入口参数传入了任务句柄和任务名，但任务名很可能在溢出时已经遭到破坏。

栈溢出钩子函数还可以在中断的上下文中进行调用。

某些微控制器在检测到内存访问错误时会产生错误异常，很可能在内核调用栈溢出钩子函数之前就触发了错误异常中断。

运行时栈侦测 —— 方法 1

当 configCHECK_FOR_STACK_OVERFLOW 设置为 1 时选用方法 1。

任务被交换出去的时候，该任务的整个上下文被保存到它自己的栈空间中。这时任务栈的使用应当达到了一个峰值。当 configCHECK_FOR_STACK_OVERFLOW 设为 1 时，内核会在任务上下文保存后检查栈指针是否还指向有效栈空间。一旦检测到栈指针的指向已经超出任务栈的有效范围，栈溢出钩子函数就会被调用。

方法 1 具有较快的执行速度，但栈溢出有可能发生在两次上下文保存之间，这种情况不会被侦测到。

运行时栈侦测 —— 方法 2

将 configCHECK_FOR_STACK_OVERFLOW 设为 2 就可以选用方法 2。方法 2 在方法 1 的基础上进行了一些补充。

当创建任务时，任务栈空间中就预置了一个标记。方法 2 会检查任务栈的最后 20 个字节，查看预置在这里的标记数据是否被覆盖。如果最后 20 个字节的标记数据与预设值不同，则栈溢出钩子函数就会被调用。

方法 2 没有方法 1 的执行速度快，但测试仅仅 20 个字节相对来说也是很快的。这种方法应该可以侦测到任何时候发生的栈溢出，虽然理论上还是有可能漏掉一些情况，但这些情况几乎是不可能发生的。

6.3 其它常见错误

问题现象：在一个 Demo 应用程序中增加了一个简单的任务，导致应用程序崩溃

任务创建时需要在内存堆中分配空间。许多 Demo 应用程序定义的堆空间大小只够用于创建 Demo 任务——所以当任务创建完成后，就没有足够的剩余空间来增加其它的任务，队列或信号量。

空闲任务是在 vTaskStartScheduler()调用中自动创建的。如果由于内存不足而无法创建空闲任务，vTaskStartScheduler()会直接返回。在调用 vTaskStartScheduler()后加上一条空循环[for(;;)]可以使这种错误更加容易调试。

如果要添加更多的任务，可以增加内存堆空间大小，或是删掉一些已存在的 Demo 任务。

问题现象：在中断中调用一个 API 函数，导致应用程序崩溃

除了具有后缀为"FromISR"函数名的 API 函数，千万不要在中断服务例程中调用其它 API 函数。

问题现象：有时候应用程序会在中断服务例程中崩溃

需要做的第一件事是检查中断是否导致了栈溢出。

在不同的移植平台和不同的编译器上，中断的定义和使用方法是不尽相同的——所以，需要做的第二件事是检查在中断服务例程中使用的语法，宏和调用约定是否符合 Demo 程序的文档描述，以及是否和 Demp 程序中提供的中断服务例程范例相同。

如果应用程序工作在 Cotex M3 上，需要确定给中断指派优先级时，使用低优先级号数值表示逻辑上的高优先级中断，因为这种方式不太直观，所以很容易被忘记。一个比较常见的错误就是，在优先级高于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断中调用了 FreeRTOS API 函数。

问题现象：在启动第一个任务时，调度器就崩溃了

如果使用的是 ARM7，那么请确定调用 vTaskStartScheduler()时处理器处于管理模式(Supervisor mode)。最简单的方式就是在 main()之前的 C 启动态码中将处理器设置为管理模式。ARM7 的 Demo 应用程序就是这么做的。

如果处理器不在管理模式下，调度器是无法启动的。

问题现象：临界区无法正确嵌套

除了 taskENTER_CRITICA()和 taskEXIT_CRITICAL()，千万不要在其它地方修改控制器的中断使能位或优先级标志。这两个宏维护了一个嵌套深度计数，所以只有当所有的嵌套调用都退出后计数值才会为 0，也才会使能中断。

问题现象：在调度器启动前应用程序就崩溃了

如果一个中断会产生上下文切换，则这个中断不能在调度器启动之前使能。这同样适用于那些需要读写队列或信号量的中断。在调度器启动之前，不能进行上下文切换。

还有一些 API 函数不能在调度器启动之前调用。在调用 vTaskStartScheduler()之前，最好是限定只使用创建任务，队列和信号量的 API 函数。

问题现象：在调度器挂起时调用 API 函数，导致应用程序崩溃

调用 vTaskSuspendAll()使得调度器挂起，而唤醒调度器调用 xTaskResumeAll()。千万不要在调度器挂起时调用其它 API 函数。

问题现象：函数原型 pxPortInitialiseStack()导致编译失败

每种移植都需要定义一个对应的宏，以把正确的内核头文件加入到工程中。如果编译函数原型 pxPortInitialiseStack()时出错，这种现象基本上可以确定是因为没有正确定义相应的宏。请参见附录 4 以获得更多信息。

可以基本相应平台的 Demo 工程建立新的应用程序。这种方式就不用担心没有包含正确的文件，也不必担心没有正确地配置编译器选项。

APPENDIX 1: BUILDING THE EXAMPLES

This book presents numerous examples – the source code for which is provided in an accompanying .zip file along with project files that can be opened and built from within the free Open Watcom IDE. The resultant executables can then be executed either within a Windows command terminal or alternatively under the free DOSBox DOS emulator. See http://www.openwatcom.org and http://www.dosbox.com for tool downloads.

Ensure to include the 16bit DOS target options when installing the Open Watcom compiler!

The Open Watcom project files are all called RTOSDemo.wpj and can be located in the Examples\Example0nn directories, where 'nn' is the example number.

DOS is far from an ideal target for FreeRTOS and the example applications will not run with true real time characteristics. DOS is used simply because it allows users to experiment with the examples without first having to invest in specialist hardware or tools.

Please note the Open Watcom debugger will allow interrupts to execute between step operations – even when stepping through code that is within a critical section. This unfortunately makes it impossible to step through the context switch process.

It is best to run the generated executables from a command prompt rather than from within the Open Watcom IDE.

APPENDIX 2: THE DEMO APPLICATIONS

Each official FreeRTOS port comes with a demo application that should build without any errors or warnings being generated[5]. The demo application has several purposes:

1. To provide an example of a working and pre-configured project with the correct files included and the correct compiler options set.

2. To allow 'out of the box' experimentation with minimal setup or prior knowledge.

3. To demonstrate the FreeRTOS API.

4. As a base from which real applications can be created.

Each demo project is located in a unique directory under the Demo directory (see APPENDIX 3:). The directory name will indicate the port that the demo project relates to.

Every demo application also comes with a documentation page that is hosted on the FreeRTOS.org WEB site. The documentation page includes information on locating individual demo applications in FreeRTOS directory structure.

All the demo projects create tasks that are defined in source files that are located in the Demo\Common directory tree. Most use files from the Demo\Common\Minimal directory.

A file called main.c is included in each project. This contains the main() function, from where all the demo application tasks are created. See the comments within the individual main.c files for more information on what a specific demo application does.

[5] This is the ideal scenario, and is normally the case, but is dependent on the version of the compiler being used to build the demo. Upgraded compilers can sometimes generate warnings where their predecessors didn't.

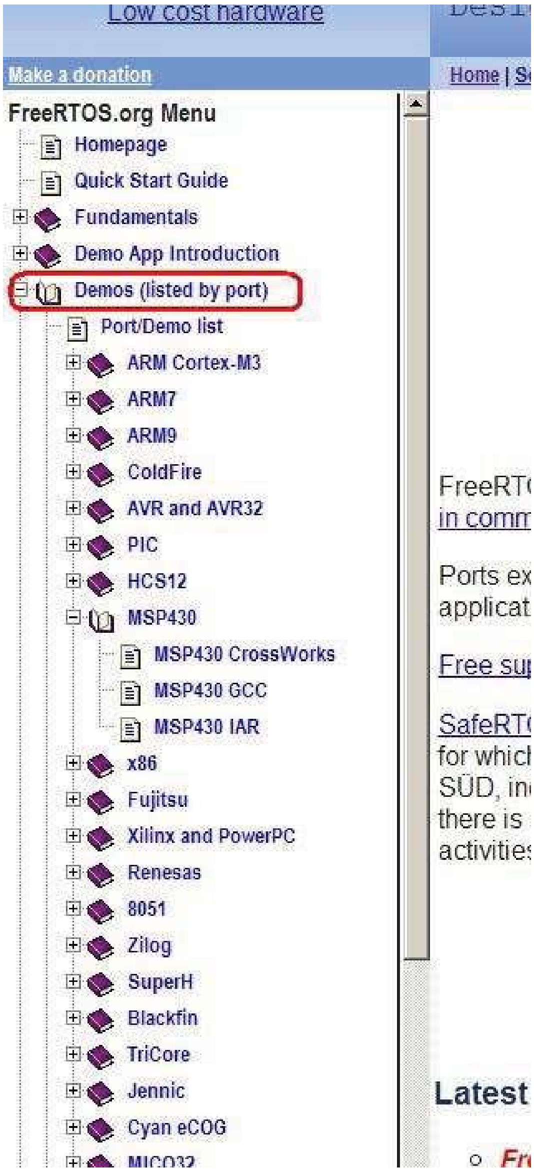


Figure 44 Locating the demo application documentation in the menu frame of the FreeRTOS.org WEB site

APPENDIX 3: FREERTOS FILES AND DIRECTORIES

The directory structure described in this appendix relates only to the .zip file that can be downloaded from the FreeRTOS.org WEB site. The examples that come with this book use a slightly different organization.

FreeRTOS is downloaded as a single .zip file that contains:

- The core FreeRTOS source code. This is the code that is common to all ports.
- A port layer for each microcontroller and compiler combination that is supported.
- A project file or makefile to build a demo application for each microcontroller and compiler combination that is supported.
- A set of demo tasks that are common to each demo application. These demo tasks are referenced from the port specific demo projects.

The .zip file has two top level directories, one called Source and the other called Demo. The Source directory tree contains the entire FreeRTOS kernel implementation – both the common components and the port specific components. The Demo directory tree contains just the demo application project files and the source files that define the demo tasks.

```
FreeRTOS
    |
    +-Demo       Contains the demo application source and projects.
    |
    +-Source     Contains the implementation of the real time kernel.
```

Figure 45 The top level directories – Source and Demo

The core FreeRTOS source code is contained in just three C files that are common to all the microcontroller ports. These are called queue.c, tasks.c and list.c, and can be located directly under the Source directory. The port specific files are located within the Portable directory tree, which is also located directly within the Source directory.

A fourth optional source file called croutine.c implements the FreeRTOS co-routine functionality. It only needs to be included in the build if co-routines are actually going to be used.

```
FreeRTOS
    |
    +-Demo          Contains the demo application source and projects.
    |
    +-Source        Contains the implementation of the real time kernel.
        |
        +-tasks.c   One of the three core kernel files.
        +-queue.c   One of the three core kernel files.
        +-list.c    One of the three core kernel files.
        +-portable  The sub-directory that contains all the port specific files.
```

Figure 46 The three core files that implement the FreeRTOS kernel

Removing Unused Files

The main FreeRTOS .zip file includes the files for all the ports and all the demo applications so contains many more files than are required to use any one port. The demo application project or makefile that accompanies the port being used can be used as a reference to which files are required and which can be deleted.

The 'portable layer' is the code that tailors the FreeRTOS kernel to a particular compiler and architecture. The portable layer source files are located within the FreeRTOS\Source\Portable\[compiler]\[architecture] directory, where [compiler] is the tool chain being used and [architecture] is the microcontroller variant being used.

- All the sub-directories under FreeRTOS\Source\Portable that do not relate to the tool chain being used can be deleted **except** the directory FreeRTOS\Source\Portable\MemMang.

- All the sub-directories under FreeRTOS\Source\Portable\[compiler] that do not relate to the microcontroller variant being used can be deleted.

- All the sub-directories under FreeRTOS\Demo that do not relate to the demo application being used can be deleted **except** the FreeRTOS\Demo\Common directory, which contains files that are referenced from all the demo applications.

  FreeRTOS\Demo\Common contains many more files than are referenced from any one demo application so this directory can also be thinned out if desired.

APPENDIX 4: CREATING A FREERTOS PROJECT

Adapting One of the Supplied Demo Projects

Every official FreeRTOS port comes with a pre-configured demo application that should build without any errors or warnings (see APPENDIX 2:). It is recommended that new projects are created by adapting one of these existing projects. This way the project will include the correct files and have the correct compiler options set.

To start a new application from an existing demo project:

1. Open up the supplied demo project and ensure it builds and executes as expected.

2. Strip out the source files that define the demo tasks. Any file that is located within the Demo\Common directory tree can be removed from the project file or makefile.

3. Delete all the functions within main.c other than prvSetupHardware().

4. Ensure configUSE_IDLE_HOOK, configUSE_TICK_HOOK and configCHECK_FOR_STACK_OVERFLOW are all set to 0 within FreeRTOSConfig.h. This will prevent the linker looking for any hook functions. Hook functions can be added later if required.

5. Create a new main() function from the template shown in Listing 77.

6. Check that the project still builds.

Following these steps will provide a project that includes the FreeRTOS source files but does not define any functionality.

```
int main( void )
{
    /* Perform any hardware setup necessary. */
    prvSetupHardware();

    /* --- APPLICATION TASKS CAN BE CREATED HERE --- */

    /* Start the created tasks running. */
    vTaskStartScheduler();

    /* Execution will only reach here if there was insufficient heap to
    start the scheduler. */
    for( ;; );
    return 0;
}
```

Listing 77 The template for a new main() function

Creating a New Project from Scratch

As just mentioned, it is recommended that new projects are created from the existing demo projects. If for some reason this is not desirable then a new project can be created by using the following steps:

1. Create a new empty project file or makefile using your chosen tool chain.

2. Add the files detailed within Table 21 to the newly created project or makefile.

3. Copy an existing FreeRTOSConfig.h file into the project directory.

4. Add both the project directory and FreeRTOS\Source\include to the path the project will search to locate header files.

5. Copy the compiler settings from the relevant demo project or makefile. In particular every port requires a macro to be set that ensures the correct kernel header files are included in the build. For example, builds targeted at the MegaAVR using the IAR compiler require IAR_MEGA_AVR to be defined, and builds targeted at the ARM Cortex M3 using the GCC compiler require GCC_ARMCM3 to be defined. The definitions get used by FreeRTOS\Source\include\portable.h – which can be inspected if it is not obvious which definition is required.

Table 21 FreeRTOS source files to include in the project

File	Location
tasks.c	FreeRTOS\Source
queue.c	FreeRTOS\Source
list.c	FreeRTOS\Source
port.c	FreeRTOS\Source\portable\[compiler]\[architecture] where [compiler] is the compiler being used and [architecture] is the microcontroller variant being used.
port.x	Some ports also require the project to include an assembly file. The file will be located in the same directory as port.c. The file name extension will depend on the tool chain being used – x should be replaced with the real file name extension.

Header Files

A source file that uses the FreeRTOS API must include “FreeRTOS.h”, then the header file that contains the prototype for the API function being used – either “task.h”, “queue.h” or “semphr.h”.

APPENDIX 5: DATA TYPES AND CODING STYLE GUIDE

Data Types

Each port of FreeRTOS has a unique portable.h header file in which is defined a set of macros that detail the data types that are used. All the FreeRTOS source code and demo applications use these macro definitions rather than directly using base C data types – but there is absolutely no reason why applications that use FreeRTOS need to do the same. Application writers can substitute the real data types for each of the macros defined within Table 22.

Table 22 Data types used by FreeRTOS

Macro or typedef used	Actual type
portCHAR	char
portSHORT	short
portLONG	long
portTickType	This is used to store the tick count and specify block times. portTickType can be either an unsigned 16bit type or an unsigned 32bit type depending on the setting of configUSE_16_BIT_TICKS within FreeRTOSConfig.h. Using a 16bit type can greatly improve efficiency on 8 and 16bit architectures but severely limits the range of block times that can specified. It would not make sense to use a 16bit type on a 32bit architecture.
portBASE_TYPE	This will be defined to be the most efficient type for the architecture. Typically this would be a 32bit type on a 32bit architecture, a 16bit type on a 16bit architecture, and an 8 bit type on an 8bit architectures. portBASE_TYPE is generally used for return types that can only take a very limited range of values and for Booleans.

Some compilers make all unqualified char variables unsigned, while others make them signed. For this reason the FreeRTOS source code explicitly qualifies every use of portCHAR with either signed or unsigned.

int types are never used – only long and short.

Variable Names

Variables are pre-fixed with their type. ‘c’ for char, ‘s’ for short, ‘l’ for long and ‘x’ for portBASE_TYPE and any other type (structures, task handles, queue handles, etc.).

If a variable is unsigned it is also prefixed with a ‘u’. If a variable is a pointer it is also prefixed with a ‘p’. Therefore a variable of type unsigned char will be prefixed with ‘uc’, and a variable of type pointer to char will be prefixed ‘pc’.

Function Names

Functions are prefixed with both the type they return and the file they are defined within. For example:

- vTaskPrioritySet() returns a void and is defined within task.c.
- xQueueReceive() returns a variable of type portBASE_TYPE and is defined within queue.c.
- vSemaphoreCreateBinary() returns a void and is defined within semphr.h.

File scope (private) functions are prefixed with ‘prv’.

Formatting

1 tab is always set to equal 4 spaces.

Macro Names

Most macros are written in capitals and prefixed with lower case letters that indicate where the macro is defined. Table 23 provides a list of prefixes.

Table 23 Macro prefixes

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Note that the semaphore API is written almost entirely as a set of macros but follows the function naming convention rather than the macro naming convention.

The macros defined in Table 24 are used throughout the FreeRTOS source.

Table 24 Common macro definitions

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

Rationale for Excessive Type Casting

The FreeRTOS source code can be compiled with a lot of different compilers, all of which have different quirks as to how and when they generate warnings. In particular different compilers want casting to be used in different ways. As a result the FreeRTOS source code contains more type casting than would normally be warranted.

APPENDIX 6: LICENSING INFORMATION

FreeRTOS is licensed under a *modified* version of the GNU General Public License (GPL) and *can* be used in commercial applications under that license. An alternative and optional commercial license is also available if:

- You cannot fulfill the requirements stated in the "Open Source Modified GPL license" column of Table 25.
- You wish to receive direct technical support.
- You wish to have assistance with your development.
- You require guarantees and indemnification.

Table 25 Open Source Vs Commercial License Comparison

	Open source modified GPL license	Commercial license
Is it free?	Yes	No
Can I use it in a commercial application?	Yes	Yes
Is it royalty free?	Yes	Yes
Do I have to open source my application code?	No	No
Do I have to open source my changes to the FreeRTOS kernel?	Yes	No
Do I have to document that my product uses FreeRTOS.	Yes	No
Do I have to offer to provide the FreeRTOS source code to users of my application?	Yes (a WEB link to the FreeRTOS.org site is normally sufficient)	No
Can I receive support on a commercial basis?	No	Yes
Are any legal guarantees provided?	No	Yes

Open Source License Details

The FreeRTOS source code is licensed under version 2 of the GNU General Public License (GPL) *with an exception*. The full text of the GPL is available at http://www.freertos.org/license.txt. The text of the exception is provided below.

The exception permits the source code of applications that use FreeRTOS solely through the API published on the FreeRTOS.org WEB site to remain closed source, thus permitting the use of FreeRTOS in commercial applications without necessitating that the entire application be open sourced. The exception can only be used if you wish to combine FreeRTOS with a proprietary product and you comply with the terms stated in the exception itself.

GPL Exception Text

Note the exception text is subject to change. Consult the FreeRTOS.org WEB site for the most up to date version.

Clause 1

Linking FreeRTOS statically or dynamically with other modules is making a combined work based on FreeRTOS. Thus, the terms and conditions of the GNU General Public License cover the whole combination.

As a special exception, the copyright holder of FreeRTOS gives you permission to link FreeRTOS with independent modules that communicate with FreeRTOS solely through the FreeRTOS API interface, regardless of the license terms of these independent modules, and to copy and distribute the resulting combined work under terms of your choice, provided that:

1. *Every copy of the combined work is accompanied by a written statement that details to the recipient the version of FreeRTOS used and an offer by yourself to provide the FreeRTOS source code should the recipient request it.*

2. *The combined work is not itself an RTOS, scheduler, kernel or related product.*

3. *The combined work is not itself a library intended for linking into other software applications.*

Any FreeRTOS source code, whether modified or in it's original release form, or whether in whole or in part, can only be distributed by you under the terms of the GNU General Public License plus this exception. An independent module is a module which is not derived from or based on FreeRTOS.

Clause 2

FreeRTOS.org may not be used for any competitive or comparative purpose, including the publication of any form of run time or compile time metric, without the express permission of Richard Barry (this is the norm within the industry and is intended to ensure information accuracy).

INDEX

Z